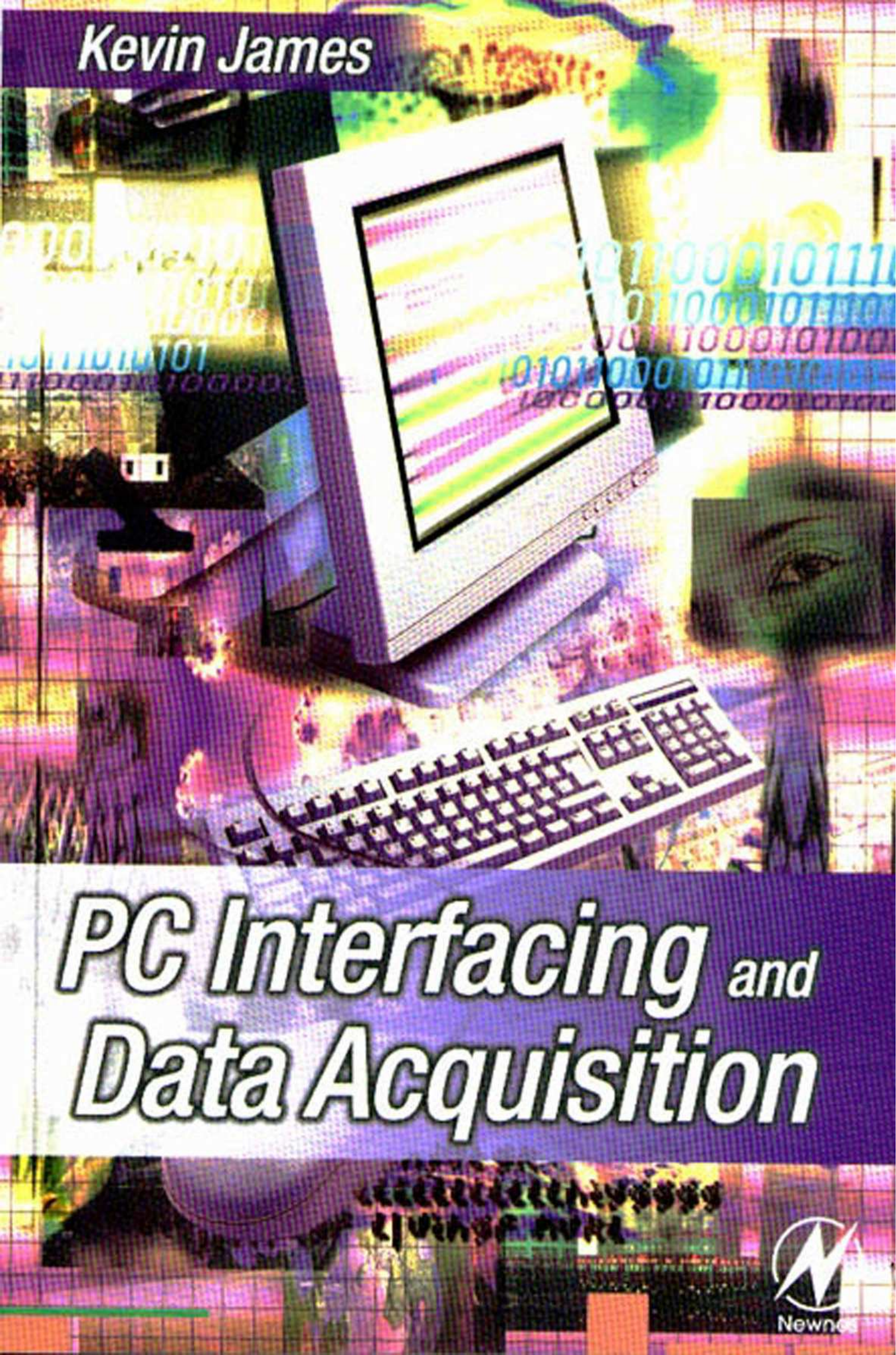
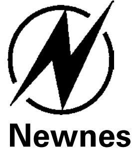

Kevin James
PC Interfacing and Data Acquisition
Newnes

PC Interfacing and Data Acquisition

This Page Intentionally Left Blank

PC Interfacing and Data Acquisition: Techniques for Measurement, Instrumentation and Control

Kevin James



Newnes
OXFORD AUCKLAND BOSTON JOHANNESBURG MELBOURNE NEW DELHI

Newnes
An imprint of Butterworth-Heinemann
Linacre House, Jordan Hill, Oxford OX2 8DP
225 Wildwood Avenue, Woburn, MA 01801-2041
A division of Reed Educational and Professional Publishing Ltd

A member of the Reed Elsevier plc group

First published 2000

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

ISBN 0 7506 4624 1

Typeset by Laser Words, Madras, India
Printed and bound in Great Britain



Contents

Part 3: I/O Techniques and Buses

Part 4: Interpreting and Using Acquired Data

This Page Intentionally Left Blank

Preface

Until fairly recently most scientific data-gathering systems and industrial control procedures were based on electromechanical devices such as chart recorders and analogue gauges. The capability to process and analyse data was rather limited (and in some cases error prone) unless one had access to a minicomputer or mainframe. Today, that situation has changed considerably. I am sure that most potential readers of this book will be aware of the profound effect the PC has had on the way in which engineers and scientists are able to approach data-gathering tasks.

Despite the now widespread use of various types of PC for automated data capture, there has been only a small number of publications on PC-based DA&C. Most if not all of these texts have concentrated on the hardware aspects of interfacing and measurement. A book emphasizing the design of DA&C *software* is long overdue.

One of the reasons for this has become increasingly apparent to me during the course of writing the present text. The subject spans numerous conventional disciplines and no single book can really do full justice to every aspect of this interdisciplinary subject. DA&C programming tends to require skills in (or at least a basic knowledge of) a range of subjects and, for this reason, the book draws together elements of programming, PC architecture, operating systems, interfacing, communications, sampling theory and process control.

My task has been complicated because of the wide range of backgrounds from which DA&C programmers tend to originate. Amongst the readership there will, no doubt, be fairly experienced programmers as well as engineers and scientists whose main area of expertise lies in fields other than computer programming. Some readers will already have a sound knowledge of data acquisition, while for others the principles of interfacing, measurement and control will be relatively new. With such a broad spectrum of potential

readers, it is inevitable that some users of the book will find that certain chapters provide unnecessary detail or that some topics are presented too concisely.

I have not assumed that the reader possesses any particular range of skills, although a broadly numerate or technical background and a basic knowledge of computer programming will undoubtedly be of benefit.

I have attempted to ensure that all information provided is correct and unambiguous. However, it is possible that a few minor errors will have found their way into the text. Unfortunately, it is in the nature of DA&C software that minor errors can have catastrophic results and, for this reason, I strongly advise you to cross-check all critical information that you use in your software against independent sources, and to thoroughly test all programs before 'going live'. I would greatly appreciate hearing of any errors in the text, whether technical or typographic. I can be contacted at: kjames_sd@hotmail.com.

A note on software examples

The code examples are presented with the primary intention of conveying the ideas presented in the text. In some cases this involves a trade-off between clarity and execution speed. In most instances I have favoured the former. You may wish to recode some of the examples to improve their efficiency and speed.

Note that the software listings are intended only as examples of how one might go about solving isolated coding problems. They are not intended as complete working programs or solutions to specific problems. For reasons of clarity, the examples are designed to operate in a real-mode (DOS) environment. In many cases the code may be adapted for use in protected mode or under 32-bit multitasking operating systems such as Microsoft Windows NT.

Although I have tested every example and they work correctly under my test conditions, factors such as execution speed and timing, hardware variability, and incompatibilities with other software (e.g. operating systems) may affect them. If you use them in your own programs you should thoroughly test them to ensure that they work correctly and reliably within the context of your application.

The examples are presented in a mixture of C and assembly language. While assembly language is *essential* for some low level programming tasks, the programmer has more scope when choosing a high level language (HLL). I have chosen C (specifically Borland C version 3) for the examples in this book mainly because it is the most widely used language in DA&C and interfacing applications.

I recognize that C code does not have a favourable reputation for clarity. For this reason, and to enable readers to translate easily to other languages, I have avoided C's shorthand notation and have used only constructs which have analogues in other HLLs. You should bear in mind that there tends to be subtle variations between different dialects of C. One such variation occurs in the various I/O instructions as described in Chapter 6. Another that is particularly relevant here concerns integer data types. Throughout the text, I have used the `int` data type as a 16-bit quantity, but in some 32-bit compilers (e.g. Microsoft Visual C++ version 4.0) it is treated as a 32-bit integer. Be sure that you know how your system interprets `int` declarations. Those readers who have any doubts over the meaning of C data declarations and statements should consult one of the numerous introductory C texts as well as their C compiler's programming manual.

This Page Intentionally Left Blank

Part 1 Introduction to Data Acquisition on the PC

This Page Intentionally Left Blank

1 The PC as a platform for data acquisition

The field of data acquisition and control (DA&C) encompasses a very wide range of activities. At its simplest level, it involves reading electrical signals into a computer from some form of sensor. These signals may represent the state of a physical process, such as the position and orientation of machine tools, the temperature of a furnace or the size and shape of a manufactured component. The acquired data may have to be stored, printed or displayed. Often the data have to be analysed or processed in some way in order to generate further signals for controlling external equipment or for interfacing to other computers. This may involve manipulating only static readings, but it is also frequently necessary to deal with time-varying signals as well.

Some systems may require data to be gathered slowly, over time spans of many days or weeks. Other will necessitate short bursts of very high speed data acquisition – perhaps at rates of up to several thousand readings per second. The dynamic nature of many DA&C applications is a fundamental consideration which we will repeatedly return to in this book.

The IBM PC is, unfortunately, not an ideal platform for DA&C. There are a number of problems associated with using it in situations which demand guaranteed response times. However, it *is* used widely for laboratory automation, industrial monitoring and control, as well as in a variety of other time-critical applications. So why is it so popular?

The most obvious reason is, of course, that the proliferation of office desktop systems, running word processing, accounting, DTP, graphics, CAD and many other types of software, has led IBM and numerous independent PC-clone manufacturers to develop ever more powerful and inexpensive computer systems. The technology is now well developed and stable in most respects. For the same reason, an enormous software base now exists for this platform. This includes all manner of scientific, statistical analysis, mathematical and

engineering packages that may be used to analyse acquired data. A wide range of software development tools, libraries, data-acquisition hardware and technical documentation is also available. Perhaps the most important reason for using the PC for data acquisition and control is that there is now a large and expanding pool of programmers, engineers and scientists who are familiar with the PC. Indeed it is quite likely that many of these personnel will have learnt how to program on an IBM PC or PC clone.

This book sets out to present some of the basic concepts of DA&C programming from a practical perspective and to illustrate how elements of the PC architecture can be employed in DA&C systems. Although it contains quite detailed descriptions of certain elements of the PC's hardware and interface adaptors, the text concentrates on the software techniques that are required to make effective use of the PC for DA&C. The first two chapters begin by discussing the structure of DA&C systems and attempt to assess how well the PC and its operating systems meet the stringent requirements of data acquisition and real-time operation.

1.1 Types of PC

Since the first models of the IBM Personal Computer (PC) were introduced in the early 1980s there have been many variants issued by IBM and by numerous 'clone' manufacturers. Each new variant has tended to introduce improved components or subsystems which enhance speed or provide some other system capability. We will not describe the various models of PC in detail here as most readers will already be familiar with the basic differences between the XT, AT, PS/2 and EISA machines. It is sufficient to note that the basic architecture of most types of PC is very similar. The differences in performance between systems arise from the different types of processor, memory subsystem and expansion bus used. These are perhaps the most important considerations although other components, such as the disk and video subsystems, can substantially affect throughput.

The IBM PC was originally developed as a stand-alone machine for office desktop use. While many DA&C applications can, and do, run successfully on such systems, desktop models do not always provide the required degree of robustness for use in harsh environments. This has led a number of manufacturers to produce more rugged versions of the PC. Many systems are built into rack-mounted chassis. They may incorporate conventional motherboard designs or they may utilize a backplane system into which a processor card, video adaptors and disk drive controllers are inserted. Ruggedized industrial PCs offer benefits such as sealed keyboards, positively

pressurized cooling systems, and anti-vibration shock mountings. Both hard disks and floppy disk drives tend to be easily damaged by dust, vibration and magnetic fields. These problems are circumvented in some systems by substituting a solid state (i.e. EPROM or SRAM based) disk emulation card which is generally less susceptible to damage.

Some industrial PCs may possess interfaces for disks, serial ports, parallel ports, and other peripheral devices on the same circuit board. Single-board computers are often integrated into dedicated equipment which is used, for example, in industrial or medical monitoring applications. These embedded systems are normally designed so as to minimize size, power consumption and cooling requirements. In these systems, hard disks are frequently replaced by ROM-based devices which provide storage for all software, including the operating system. Embedded PC controllers are also used in mobile equipment. However, there are a number of other options when it comes to mobile computing. There are now many notebook PCs and ruggedized portable computers on the market. These can easily interface to external data logging or control equipment in order to facilitate configuration or downloading of acquired data.

Ruggedized PCs, embedded PC systems, portable machines and desktop PCs all share the same basic architecture and are generally capable of running the same software. The structural differences between them are largely irrelevant to the software engineer. Indeed software can usually be developed on a desktop system and then transferred to a ruggedized or portable PC without modification, although minor changes may sometimes be needed when porting to embedded systems in order to accommodate ROM-based operating systems or to interface to specialized external buses.

1.2 The processor

Most readers of this book will already be aware of the different types of processor and coprocessor used in the PC range. This section summarizes the most important characteristics of each of the main classes of processor. The text by Hummel (1992) provides more detailed descriptions of the various processors and coprocessors available.

The 80x86 family of processors

Pentium processors are perhaps the most recognized components of today's PCs. They originate from a long line of Intel processors dating back to the 1970s (see Table 1.1). The capabilities of the

Table 1.1 *Comparison of 80x86/Pentium processors*

Processor	*Address range*	*Data width (bits)*	*Clock (internal) (MHz)*	*Approx. relative speed*[3]	*New features and notes*
8088	1 MB	8	4.77	1	Real mode only.
8086	1 MB	16	4.77, 8	1.5	Real mode only. Required 8087 floating-point unit.
80286	16 MB	16	6–16	5	Limited protection features in protected mode. Required 80287 floating-point unit.
80386SX	16 MB	32[1]	16–25	10	Enhanced protected V86 mode. Required 80387 floating-point unit.
80386DX	4 GB	32	16–40	15	32-bit data and address buses. Required 80387 floating-point unit.
80486SX	4 GB	32	25–40	40	Parallel instruction execution. 8 Kbyte on-chip cache. Internal clock doubling, tripling and quadrupling circuits. Required 80487 floating-point unit.
80486DX	4 GB	32	25–100	60	On-chip numeric processor.
Pentium	4 GB	32[2]	60–166	200	Dual execution pipeline. Enhanced branch prediction. Enhanced V86 paging. Multiprocessor support.
Pentium Pro	64 GB	32[2]	200, 266	500	Triple pipelining. 256 Kbyte L2 cache. 36-bit address bus.
Pentium II	64 GB	32[2]	200–450	800	Enhanced L1 and L2 caches. Power saving features. MMX extensions.
Pentium III	64 GB	32[2]	500+	1000+	Very efficient floating-point unit. Katmai New Instructions and new KNI mode.

[1] 16-bit external bus.
[2] 64-bit external bus.
[3] Integer processing. Figures are a rough guide only. Actual speed depends on clock rate, instruction mix and performance of PC subsystems.

earlier processors will be of little relevance to most readers who, nowadays, are not likely to encounter anything more primitive than an 80486. For this reason we will not discuss them in any further detail. We should remember, though, that some specialized systems (particularly embedded PC applications) still make use of the earlier 8086, 80286 and 80386 processors. Indeed, special versions have been developed for this market. The 80186, for example, is similar to the 8086, but also possesses on-chip DMA (Direct Memory Access) and interrupt controllers and other support circuitry. The 80186 and similar special-purpose processors are not used in a normal PC.

From the viewpoint of application-software development, it is convenient to divide the various PC processors into three classes: real-mode processors (8088, 8086 and compatibles such as the NEC V20 and V30); the intermediate 80286 processor (which we will not discuss); and full 32-bit processors (80386, 80486, Pentiums and Celeron processors).

In essence the early real-mode processors (used on the first models of PC) ran only one program at a time, provided limited memory addressing (up to 1 MB), and operated relatively slowly (being clocked at 4.77 to 10 MHz, typically).

At the other extreme, the 80486DX and Pentium class processors can address large amounts of memory (4 GB), and possess features for task switching, high speed processing and memory/hardware protection. These capabilities are used by sophisticated 32-bit operating systems such as Windows NT to implement efficient multitasking and to control access to system resources.

Intel released a cheaper alternative to the Pentium in 1998: the Celeron processor. This is similar to the Pentium II, but without the latter's built-in level 2 cache. Despite the fact that, by most standards, the Celeron is significantly slower, it is becoming popular in some industrial applications, particularly in embedded systems.

Pentium II processors operate at up to 450 MHz internally. This and enhancements such as 64-bit external data bus, separate caches for instructions and data, a much improved instruction handling capability and very efficient numeric processing are responsible for the superior performance of Pentium-based PCs. The Pentium III offers further improvements in performance. Initial versions are clocked at up to 500 MHz and faster versions will no doubt be available by the time this book is published. Floating-point performance has been enhanced in the Pentium III with the addition of a special instruction set (Katmai New Instructions, or KNI) and new registers. This provides up to about 2×10^9 floating-point operations per second (2 Gflops): sufficient for the processor to take on tasks that

might otherwise have required a specialized Digital Signal Processor (DSP): real-time audio processing, for example.

Because each new processor in the sequence incorporates a superset of the instructions and features of earlier processors, they are termed ‘backward compatible’. Software written for an 80286 processor, for example, will generally be able to run on 80386 and all later processors. Even the latest Pentium processors can operate in real mode, emulating the early 8086. Note, however, that the converse is not true: an 8086 will not run most of the software written for the Pentium. In spite of this backward compatibility, the timing of many instructions varies between processors. The speed of most instructions tends to be greater in the newer processors although some instructions may execute more slowly. This point should be borne in mind when writing very time-critical code, particularly if the software is intended to run on a range of different processors.

Processor modes

The 8086 processor is capable of directly addressing up to 1 MB of memory. It is designed to support the execution of only one program (or process) at any time. This process has complete control over the PC and has direct access to all addressable memory and I/O locations, even those belonging to the system BIOS or to the operating system itself. Because there are no protection mechanisms to prevent interference between processes it is difficult to implement safe multitasking (see Chapter 2) on the 8086. The 8086’s mode of operation is known as real address mode (often abbreviated to just ‘real mode’). All later processors support real mode as well as other modes that allow access to more than 1 MB of memory.

The protected mode available on 80286 and later processors helps to circumvent the 1 MB limitation. As well as providing access to more memory, it incorporates a number of mechanisms which help to prevent processes from conflicting with each other or with the operating system. All subsequent processors (i.e. 80386 and later) also possess a virtual 8086 (V86) mode. In this mode, the processor operates as multiple virtual 8086 machines, dividing its time between each. Programs are allocated their own virtual machine and in this way it appears to the program that it is running on its own 8086 processor. Each virtual machine may have its own DOS environment and is isolated from the rest of the system. The program running on each virtual machine believes that it has full control of the system, as on a real 8086. Interprocess memory conflicts and I/O conflicts are avoided by means of sophisticated protection mechanisms provided by the processor (as described later in this chapter). In order to

perform multitasking using the processor's protected or V86 modes the whole machine has to be managed by suitable operating system software. We will discuss this topic in Chapter 2.

Although the modes available on the more advanced processors are very efficient, their protection mechanisms can involve a substantial software overhead, especially if complex multitasking operating systems are used to mediate between processes. DA&C programs are normally relatively small and uncomplicated, and a simple real-mode environment (e.g. a DOS-based system) is often the most suitable. A protected-mode system can, however, provide the potential for a greater degree of reliability. The inherent protection mechanisms can help to prevent resource conflicts and may highlight certain types of coding error during development.

Registers

Throughout this book I will make frequent references to an important feature of the processor: its registers. The basic concepts are introduced below. However, this is only a very brief overview to aid your understanding of the examples presented in subsequent chapters. You should consult a specialist text on processor architecture or assembly language programming – e.g. Hummel (1992), Swan (1989) or Holzner and Norton (1991) – for a more detailed discussion of this subject.

Each processor in the 80x86 family possesses several 16-bit registers which are used to hold data and memory addresses. In many operations, you have a choice of which register to use. However, most registers are designed specifically for certain operations. Some registers, such as CS, DS and SS, address particular memory segments (blocks of up to 64 KB addressable in real mode). Others (e.g. IP, SP, BX) can be used to address individual bytes or words as offsets from the beginning of an associated segment. Yet other registers are used to hold numeric data. Some of the 16-bit registers (i.e. AX, BX, CX and DX) allow their high and low order bytes to be addressed separately. For example, the high order byte of AX is referenced within an assembly language program as AH, and the low order byte as AL. The AX register is used exclusively in certain operations such as reading from or writing to an I/O port. The Flags register contains various bits which indicate the results of arithmetic operations or which control how particular features of the processor operate.

The 80386 and subsequent processors are equipped with 32-bit registers. Each of the 16-bit registers mentioned above is actually implemented as the low order 16 bits of the corresponding 32-bit register. Just as it is possible to separately reference the high and low

order bytes of certain 16-bit registers, one can reference either the full 32-bit register (by preceding the normal register designation with an 'E', e.g. EAX) or only the low order 16 bits (e.g. AX). For the sake of simplicity and compatibility with the 80286 and earlier processors, only the 16-bit register set is used in the examples presented in the remainder of this book. Those readers who are unfamiliar with assembly language should consult a book such as Swan (1989) for an introduction to this subject.

The most important point to remember about the registers is that their contents completely define the state of the processor at any given time. The registers may hold a variety of information relating to the current process. This includes the address of the next instruction to be executed, intermediate results, the interrupt state and many other essential parameters. If the register contents are incorrectly modified or become corrupted it is very likely that this will result in the failure of the software. You should bear this in mind when dealing with any form of context switch such as an interrupt or task switch, and take appropriate steps to preserve the state of the registers. Refer to Chapter 2 for more on task switches and concurrent processing, or to Chapter 5 for a detailed discussion of interrupts.

Numeric processing

Predecessors of the 80486DX have a limited mathematical processing capability. While they are able to perform a variety of integer arithmetic, data transfer, and logical operations, they were not designed to undertake floating-point calculations. Many compilers and development tools incorporate floating-point software libraries. These contain long and complex routines to facilitate floating-point computation. Unfortunately, floating-point software can be slow. When many calculations have to be performed, the burden placed on the processor may unacceptably degrade the system's throughput. This problem can be particularly acute in high speed DA&C applications.

The alternative technique is to use special hardware for numeric processing. A numeric processing unit is dedicated to performing floating-point calculations and operates more or less in parallel with the main processor. It supports a number of floating-point data types and provides facilities for performing trigonometric and transcendental functions. The 80486DX and Pentium class processors have built-in numeric processing units, but earlier processors required a matching numeric coprocessor IC. This hardware solution makes very substantial increases in throughput possible, although the degree of benefit gained does, of course, depend upon the

nature of the software. Numeric processors are not essential in all DA&C applications. Many programs execute only integer instructions during the period of data acquisition. However, a numeric processor can be invaluable in applications which have to execute mathematical control algorithms (e.g. PID control) or which must undertake any form of real-time signal processing.

The presence or otherwise of a numeric processor is normally transparent to programmers working with C, Pascal or other high level languages. The programmer will normally only have to select a compiler 'switch' in order to generate code for a numeric processor or to emulate one in software. He or she need not be concerned with *how* floating-point calculations are actually performed. This is not true, however, for assembly language programmers. These readers are advised to consult more specialized texts on the subject such as Hummel (1992) or Holzner and Norton (1991).

1.3 Memory

As we have already seen, modern PCs can address up to 4 GB of memory, although most contain very much less. Figure 1.1 illustrates the PC's memory space and shows some important regions within the address map. The addressable range is processor (and mode) dependent.

When operating in real mode, the 80x86 and Pentium processors employ a segmented memory addressing scheme. Each memory address is specified in the software by the contents of a segment register and an offset register. In real mode both of these registers are 16 bits wide and thus a memory segment is defined as a memory block up to 65 536 bytes in length. A segment begins on any paragraph (16-byte) boundary. The contents of the segment and offset registers are combined to form a physical address by multiplying the contents of the segment register by 16 and then adding the result to the value held in the offset register. This generates a 20-bit address which can be used to access any location in the 1 MB memory area. The segmented memory scheme can complicate programming somewhat, although it does have a number of practical advantages. It provides a means of dividing memory up into convenient segments, the beginning of each segment being addressed by the contents of the segment register. Successive bytes within a segment can then be easily referenced by incrementing or decrementing a single 16-bit offset register.

The addressing method used in the 80286's protected mode is similar. However, the value held in the segment register no

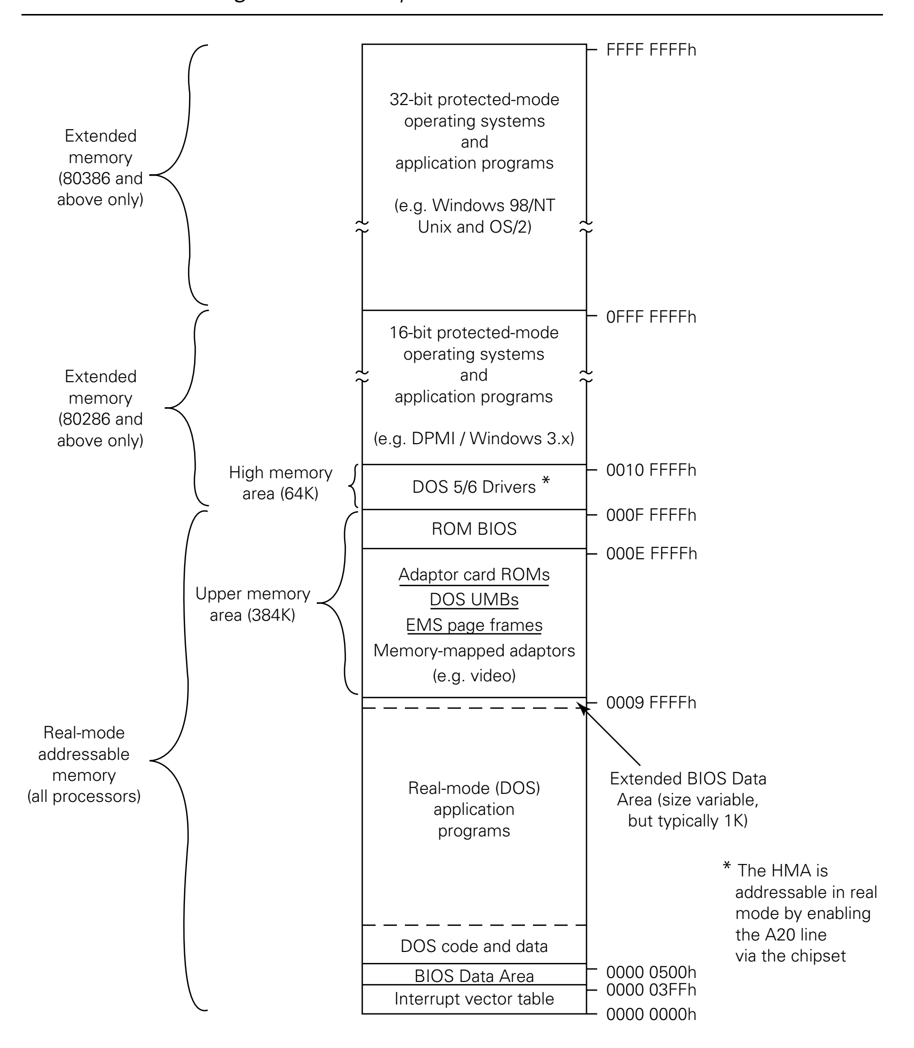


Figure 1.1 *The PC's memory map*

longer corresponds to a physical segment base address. Instead, it is used as a selector. This is a pointer to an entry in a table maintained by the operating system. Each entry in this table is known as a descriptor and specifies the physical address of the segment of memory which is to be accessed. The selector and descriptor also contain other data relating to the memory segment. This includes the information necessary for operating systems to implement interprocess protection and memory management. For example, the descriptor specifies whether the segment referenced is a code or data segment and thus provides a mechanism for the

operating system to trap actions such as inadvertent writes to a code segment. It also specifies the size of the segment so that accesses to memory beyond the segment limit can be detected. The 80286 can access up to 16 MB of memory.

A similar system is used on the 80386 and later processors when they are running in protected mode. However, these processors can use a 32-bit flat addressing scheme in which the selector is kept fixed by the operating system and the programmer addresses memory by means of only a 32-bit offset. This provides access to up to 4 GB of memory. The 80386 and later processors also provide an additional memory management facility, known as *paging*. When paging is disabled, the address determined from the descriptor represents the physical memory address (as in the 80286 processor). When paging is enabled, the linear (or virtual) address read from the descriptor table has to undergo another translation step in order to arrive at the physical address. The page translation mechanism makes possible the V86 mode and is also essential for a number of other advanced operations on the 80386 and later processors. Unlike the segmentation scheme, page translation is generally transparent to the applications programmer. It is normally managed invisibly by the operating system. However, the paging mechanism does have certain implications for real-time DA&C systems. It allows an operating system, such as Windows NT, Windows 95/98 (or Windows 3.1 operating in enhanced mode), to temporarily swap blocks of memory out to a hard disk. Although this can be a great advantage in non-time-critical systems it may be unacceptable in real-time DA&C applications as it has the potential to introduce variations in the time taken for the DA&C program to respond to external events.

The protected-mode segmentation scheme, the page translation mechanism and V86 modes are quite involved topics and full descriptions of them are beyond the scope of this book. You should consult a text on the subject of operating system architecture or on the processor itself (e.g. Hummel, 1992) for further information.

Accessing memory above 1 MB from real mode

Many DA&C applications are relatively straightforward and may not need the complex multitasking and protection capabilities offered by the processor's protected and V86 modes. Often, however, they do require large quantities of memory in which to store acquired data, and this is not directly available in real mode. If you prefer the simplicity, speed and degree of control offered by a real-mode DOS-based system (perhaps one of the specialized real-time versions

of DOS), there are several ways in which to gain access to memory above the 1 MB limit.

First, you could make use of two BIOS services provided on the IBM AT and compatible machines. These services allow data to be moved between real-mode-addressable memory (i.e. memory below the 1 MB boundary) and extended memory. This technique is rather slow and requires a degree of buffering in real-mode-addressable memory. It also relies upon the cooperation of all other processes running on the machine in order that they do not overwrite another's data.

The second method of accessing extended memory is to employ an extended memory driver conforming to the Extended Memory Specification (XMS). Such a driver, HIMEM.SYS, is used by Microsoft Windows 3.1 for managing extended memory. It provides a comprehensive set of services which can be used to access memory above the 1 MB boundary as well as the so-called Upper Memory Blocks (UMBs) in the 640 KB to 1 MB area.

The third method is simply to make use of a RAM disk (also known as a Virtual disk) device driver. This sets aside an area of memory (usually extended memory) to emulate a disk drive. The RAM disk operates in the same fashion as a normal hard or floppy disk. Although it is many times faster than a typical hard disk drive, data still has to be transferred via the DOS file and device driver system and so this method is generally slower than direct memory storage.

The final approach is to employ an expanded memory system. This technique is largely obsolete on the PC, but it is instructive to consider it briefly because some specialized data-acquisition hardware makes use of a similar system for transferring data to and from the PC's memory. Expanded memory has been used in embedded systems for some time, and a number of 8086-compatible processors that have been developed especially for embedded applications include on-chip expanded memory support.

Expanded memory is essentially bank switched memory which can be selectively paged in and out of a memory window (known as a page frame) residing below the 1 MB real-mode-address limit. Data may be read from, or written to, expanded memory through this window as though one were accessing the PC's memory. The DA&C program can select new pages at any time by calling a group of system services that are provided by an expanded memory device driver. The services generally conform to a standard known as the Expanded Memory Specification (EMS). Versions 3.2 and 4.0 of this standard are the most widely used. One of the more effective EMS implementations utilizes the paging facilities provided by the 80386

and later processors, allowing some or all of the PC's extended memory (i.e. that above 1 MB) to be treated as expanded memory.

Although the bank switching and paging mechanisms used on the PC are fast and ideally suited to DA&C, they have to be managed by some form of device driver. As with all drivers and programs written by third parties, you should be sure that they do not compromise the deterministic qualities necessary in real-time systems (see Chapter 2).

EMS, XMS and the extended memory BIOS services are covered in many books on the IBM PC such as Duncan (1989), Duncan *et al.* (1990) or Dettmann and Johnson (1992).

1.4 Input/output ports

In addition to its memory, the PC has another entirely separate address space. This is dedicated to transferring data to or from peripheral devices and is known as Input/Output space (or simply I/O space). Just as the PC's memory space is divided into separate byte locations, the I/O space consists of many byte-sized I/O ports. Each port is addressable in much the same way as memory, although an additional control line is used within the PC to distinguish between memory and I/O port accesses. I/O space consists of a contiguous series of I/O addresses. Unlike memory space, the I/O address space is not segmented and cannot be paged. In fact, the processor references I/O ports by means of a 16-bit address and this means that no more than 65 536 I/O ports can be supported by the PC. In practice, this is further limited by the I/O address decoding scheme used on the PC and its adaptor cards.

The I/O ports provide a means of sending data to, and receiving data from, devices such as the video adaptor, the disk subsystem, or analogue-to-digital converters (ADCs) on plug-in data-acquisition cards. Software can use the assembly language `IN` or `OUT` instructions, or their high level language counterparts, to communicate with hardware devices via the I/O ports. These are discussed in more detail in Chapter 6, but for the moment we will consider a simple example. Suppose that a plug-in 8-bit ADC card possesses control and data registers that are each mapped to one of the PC's I/O ports. The software starts the analogue-to-digital conversion process by writing a bit pattern to the I/O port that maps to the ADC card's control register. When the ADC has finished the conversion it might set a bit (known as the End of Conversion, or EOC, bit) in another register to indicate that digitized data is now available. In this way, the software is able to detect the EOC bit by reading the corresponding I/O port. Knowing that the conversion had been

completed, the software would then read the digitized data from a data register mapped to a third I/O port.

I/O port allocation

Hardware devices map their registers to specific I/O ports simply by decoding the PC's address bus and control lines. In this way, a specific combination of address and control lines is needed to cause data to be transferred from the register to the PC's data bus or vice versa. Some I/O ports can only be read or written, while others are capable of bidirectional data transfer. Whether ports are read-only (R/O), write-only (W/O) or read-write (R/W) is determined by how the hardware decodes the address and control lines. The processor itself makes no distinction between ports in this regard. You can still perform an `IN` instruction for a write-only port although the results of such an action will generally be indeterminate.

The PC and adaptor-card hardware do not fully decode the address lines. In fact, in the IBM PC, XT, AT and compatible machines, including the PS/2 line, only the lower 10 lines are used. This means that it is possible to address only 1024 separate I/O ports. Even certain addresses within *this* range are not fully decoded. Thus some devices which should require only two or three registers may actually occupy a much larger block of I/O addresses: the same registers being mirrored at a series of other addresses within the block. A much more satisfactory approach is taken on EISA systems. These decode the address lines more fully, providing additional I/O ranges that are dedicated specifically to the system motherboard or to adaptors residing in each of the EISA expansion bus slots. On each class of PC, certain I/O addresses are reserved for particular devices. Table A.3 in Appendix A provides an overview of I/O port usage and may be used as an aid to selecting ports for use by data-acquisition adaptor cards.

I/O protection mechanisms

The PC's I/O ports are always accessible in real mode. In protected and V86 modes, however, the processor can be programmed to restrict access to I/O addresses. This facility is used in multitasking operating systems such as OS/2 and Windows NT to control which processes (i.e. running programs) will be allowed to read and write the I/O ports. In this way it is possible for the operating system to mediate between two or more processes that need to access the same I/O device. The operating system runs at a high privilege level, which means that it is allowed to execute certain privileged instructions.

These include instructions that access the I/O ports and those which change the state of the processor's Interrupt Flag (see Chapter 5).

In protected and V86 modes, when a program operating at a low privilege level attempts to execute one of the privileged I/O instructions, the processor generates a General Protection exception. This causes control to be immediately passed to the operating system, which can then oversee the I/O port access. The details of this process are quite involved and cannot be covered here. You should consult a text such as Hummel (1992) for more on this topic.

One of the consequences of the I/O protection mechanism is that an application program running in protected or V86 mode (e.g. under OS/2 or Windows) will generally be prevented from *directly* accessing the I/O ports. I/O port accesses require at least some operating system intervention and this reduces the maximum possible throughput of the system. It also contributes to a degree of uncertainty in the speed at which the system will respond. This can be a particularly important consideration when designing a real-time DA&C program.

1.5 Buses and adaptor card slots

Passing data to and from a DA&C card via an I/O port actually involves transferring the data over one or more system buses. Figure 1.2 illustrates a variety of buses that can be interfaced to the PC. A typical PC may not contain all of the buses shown, although the PCI and ISA buses are present in most systems. Other types of bus (many of them proprietary systems) can be interfaced by means of special adaptors or bridges to the PC. The IEEE-488 bus and the VXI bus, for example, are used in specialized instrumentation applications. Of primary concern here though are the PC's native buses – i.e. the ones that are an integral part of the PC's own architecture.

The type of bus used within the PC not only has a bearing on the type of interface card that can be connected, it may also have a profound effect on the throughput of the system as a whole. Although normal bus operation cannot be modified under software control and is largely transparent to the programmer, it is of great importance in interfacing and so a brief overview is provided below.

The ISA bus

Until the mid-to-late 1990s, the Industry Standard Architecture (ISA) bus dominated the PC market and was the interface used for most

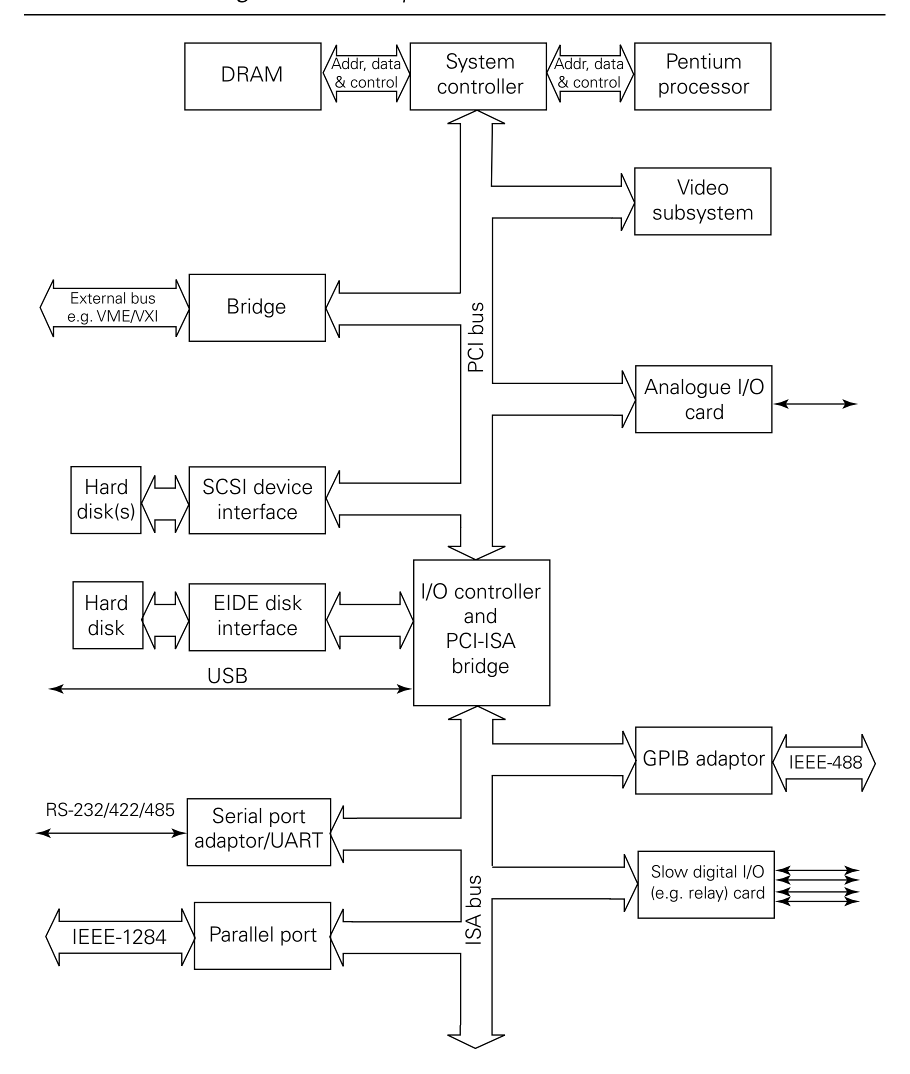


Figure 1.2 *Example bus connections and interfaces on a PC used for data acquisition. Note that not all devices and buses shown will be present on every system, and some systems will incorporate additional devices*

plug-in DA&C cards. It is derived from the earlier, and slower, 8-bit bus used in the IBM PC and XT (known as the PC bus or XT bus). Note that the 16-bit ISA bus (also known as the AT bus because it was introduced in the IBM AT computer) is in some literature also misleadingly referred to as the PC bus.

The ISA bus incorporates a number of enhancements over the XT bus, such as a 16-bit data path, a 16 MB addressing capability,

an increased number of interrupt request lines (see Chapter 5) and additional DMA channels (see Chapter 6). The extra data, address and control lines necessary to interface to ISA type adaptor cards were added in a second connector placed in line with the original XT type connector. Although a few of the connector pins on the XT connector were redesignated, the ISA bus connector provides full backward compatibility with the older XT cards. Most ISA machines are equipped with several 16-bit ISA slots and one or two 8-bit XT type slots. With a few exceptions (noted below), 8-bit cards can also be inserted in the XT portion of 16-bit ISA slots.

The ISA bus clock speed is not tied to the processor clock as it was in the XT bus. Widely differing bus and processor clock speeds are used on ISA machines and synchronization between the two is maintained by means of special support circuits. The IBM AT's bus was clocked at 8 MHz. Many newer systems allow the bus clock speed (and indeed the processor and DMA clock speeds) to be reprogrammed using a BIOS configuration utility. The chosen speed is recorded in the system's CMOS RAM. A high frequency (e.g. 10 or 11 MHz) may be selected provided that all adaptor cards will operate reliably at this speed. Most modern ISA adaptor cards are capable of running at 10 or 11 MHz, but some older DA&C cards are not.

Bear in mind that even the standard 8 MHz ISA clock speed may be incompatible with some older ADC or counter/timer cards that were intended specifically for IBM PC or XT systems. These cards are designed to provide their on-board components with clock signals derived from the PC's 4.77 MHz bus clock and are, therefore, unsuitable for use with the higher clock frequencies present on the ISA bus. Indeed they are also incompatible with the 8 or 10 MHz XT buses employed in some XT clones. Generally speaking, this is no longer a problem with modern DA&C cards as these tend to be driven from their own dedicated oscillator, rather than from the system bus clock. You should, however, be wary of this potential difficulty when using some pre-1990 DA&C cards.

Today, new desktop PCs now rarely possess more than one or two ISA card connectors, the remaining expansion capability being provided by the PCI bus, which we will discuss shortly. However, the ISA bus is far from obsolete in the industrial data-acquisition market. Many rack-mounted industrial PCs still employ this standard and there are numerous ISA bus DA&C cards still on the market. Before discussing the PCI bus, it is appropriate to briefly mention two other buses: the MCA bus and the EISA bus. Although these are both technically superior to the ISA bus in many respects, they have not enjoyed such widespread use.

The MCA bus

The MCA (Micro-Channel Architecture) bus was developed by IBM for its range of PS/2 computers. MCA was more rigidly specified than the ISA bus in terms of it physical, electrical and timing characteristics, and incorporated a software-based card configuration facility. The latter feature, called Programmable Option Select (POS), circumvented the need to use DIP switches or jumpers for selecting options such as base address or interrupt levels. As all configuration is performed via manufacturer-supplied software, the details of POS operation are rarely of interest to the DA&C programmer. Readers are referred to the text by Eggebrecht (1990) for more information on POS.

The EISA bus

The main disadvantage of the MCA bus was its incompatibility with the earlier XT and ISA buses. A consortium of PC manufacturers attempted to circumvent this problem by developing an enhanced version of the ISA bus, known as the Extended Industry Standard Architecture, or EISA, bus. This provided a number of benefits similar to those of MCA while maintaining full backward compatibility with ISA cards. EISA buses, which are used in some 80386 and later systems, incorporate a 32-bit data bus and have an enhanced slot-specific I/O addressing capability. Like MCA, EISA cards are configured by means of software utilities and data files supplied by the manufacturer.

The PCI local bus

Local buses began to emerge as potential competitors to conventional expansion buses such as ISA in the mid-1990s. Whereas conventional buses have to employ special circuitry to manage bus traffic and to synchronize high speed processors with slower bus operations, local buses are more tightly coupled to the processor.

Currently, the dominant local bus standard is Intel's PCI (Peripheral Component Interconnect) bus. Although the latest PCI standard (version 2.2) allows for 64-bit transfers at 66 MHz, standard PC-based PCI implementations currently provide a 32-bit data path. Because PCI operates at the processor's clock frequency (i.e. the frequency of the clock signal supplied to the processor, rather than the processor's internal clock frequency), it is capable of very high rates of throughput. The PCI bus also supports bus mastering in which PCI devices can take control of the bus in order to transfer

data. This is much like the DMA technique used on the ISA bus (see Chapter 6). The principal difference is that each device supplies its own bus-mastering hardware rather than relying on the PC's DMA controller. Additional performance enhancements can often be realized by this means because bus transfers can be carried out in parallel with certain processor operations. PCI devices can, for example, exchange data along the bus at the same time that the processor is accessing system memory.

Transfer rates

Table 1.2 summarizes the main characteristics of the buses discussed so far. A 32-bit PCI bus clocked at 33 MHz can, in theory, provide a data transfer rate of 132 MB/s. This represents a huge increase over conventional buses. An 8 MHz ISA bus was, for example, capable of transferring data at up to 16 MB/s. The MCA and EISA buses fare

Table 1.2 *PC expansion buses*

Bus	*Address width (bits)*	*Data width (bits)*	*Standard clock rate (MHz)*	*Max. throughout at standard clock (MB/s)*	*Notes*
PC (XT)	20	8	8	8	Six IRQ lines. Three DMA channels.
ISA (AT)	24	16	8	16	Twelve IRQ lines. Seven DMA channels.
MCA	24	32	Variable (typically 10–20)	20–160	Maximum transfer rates achieved in data streaming mode. DMA implemented via bus mastering with up to 16 arbitrating devices.
EISA	32	32	8	33	Quoted throughput achieved in data streaming mode.
PCI	32	32 or 64	33 or 66	132[1]	Intelligent bus mastering with support for DMA. Quoted transfer rate is achievable in burst mode only.[1]

[1]For a 32-bit implementation running at 32 MHz. Maximum throughput increases proportionately for faster or wider versions of PCI.

somewhat better. MCA supports 32-bit data transfers at rates up to 20 MB/s. Higher rates (typically 40 to 80 MB/s) are achievable with a special data streaming mode. EISA systems provide bus transfer rates of up to 32 MB/s. Bear in mind that these maximum transfer rates cannot always be realized in practice. Throughput is often limited by factors other than bus bandwidth.

The AT's DMA controller can provide a throughput of up to approximately 1 MB/s (or 2 MB/s, depending upon whether an 8-bit or 16-bit DMA channel is used). A greater throughput can sometimes be achieved using programmed I/O: typically up to 3 MB/s on a fast machine. In practice, however, delays inherent in other components (e.g. the ADC conversion time, multiplexer settling times, signal conditioning bandwidth – see Chapter 3) tend to be the principal throughput-limiting factors. For this reason, the maximum bus transfer rate cannot usually be realized and in many applications bus speed has only a minimal effect on the overall system throughput. DMA, programmed I/O and throughput rates are discussed in more detail in Chapter 6.

PCMCIA interface

Like local buses, PCMCIA cards (sometimes known as just PC cards) are a fairly recent innovation in PC interfacing. The PCMCIA (Personal Computer Memory Card International Association) standard defines a hardware and software interface for attaching miniature adaptor cards to the PC. It was originally intended as a standard bus for interfacing removable memory cards to portable computers, although it has now been adopted for other peripheral devices such as serial ports, modems, network interfaces and hard disks. DA&C component manufacturers now also produce data acquisition cards in PCMCIA format. At the time of writing, these devices are largely limited to simple mainstream DA&C functions (8 channel multiplexed ADCs, dual DAC cards, counter/timers, simple digital I/O facilities etc.) and provide reasonably high, although not exceptional, throughput. Few PCMCIA cards offer more advanced features such as very high speed ADCs, FIFO buffers or an on-board processing capability. A number of industrial communications PCMCIA cards (RS-232/422/485 or IEEE-488) are also available.

As mentioned above, PCMCIA cards are small: about 2 inches (50 mm) across. They are produced in various thicknesses: Type I cards are 3.3 mm thick; Type II cards are 5.0 mm thick; and Type III are 10.5 mm thick. The extra thickness of Type III cards is required principally to accommodate miniature hard disks and radio frequency communications products. DA&C cards are normally of

Type II. Most notebook PCs are able to accommodate at least two of these Type II cards, permitting moderately complex DA&C systems to be designed around a portable computer.

PCMCIA cards offer several benefits. They are software configurable, so installation (I/O address selection, interrupt selection etc.) can generally be automated. Apart from the fact that they follow a fairly rigid specification in terms of power usage, signal timing, and physical size, they also offer specific advantages for users of DA&C systems. Their 16-bit data bus provides reasonably high rates of throughput at moderate cost. Because of their size, PCMCIA cards are extremely portable and, when used in conjunction with notebook PCs, open up the possibility of data acquisition in awkward environments (e.g. in moving vehicles). They can be unplugged from the PC or from other DA&C system components, facilitating relocation from one DA&C site to another. PCMCIA cards also have a *hot insertion* capability. This permits cards to be removed from the computer and swapped for other cards without having to switch off the PC.

Due to the small size of the cards, subminiature connectors are employed. This means that PCMCIA DA&C cards normally have to be used in conjunction with extension cables and screw terminal panels which will accept the field connections from transducers or signal-conditioning units. In certain applications, these devices may also include sensor excitation references or isothermal connections for thermocouple cold-junction compensation (see Chapter 3). As the PCMCIA circuit board is fully enclosed it is difficult to gain access to trimpots or to test points for calibration or fault diagnosis. However, PCMCIA DA&C cards are normally factory calibrated where necessary and any subsequent recalibration can usually be performed by adjusting scaling factors and offsets in software (see Chapter 9). Most PCMCIA card manufacturers supply software drivers and, in many cases, configuration, calibration and diagnostics programs as well.

Industrial and instrumentation buses

As mentioned previously, the standard desktop PC format is not robust enough for use in harsh industrial environments. Industrial DA&C systems often employ ruggedized versions of the PC in specially designed rack-mounted enclosures. However, the physical properties of the enclosure are not the only consideration. The standard PC architecture may not have the interfacing support needed to directly manage some complex industrial sensing or control systems. It does, nevertheless, have many other advantages (noted in the

introduction to this chapter) which makes it highly desirable in this type of application.

A number of manufacturers have attempted to bridge the gap between the desktop PC and more robust industrial systems by producing versions of the XT, ISA or PCI buses in a passive backplane format that is suitable for use in industrial 19 inch rack-mounted enclosures. These backplanes usually have a large number of expansion slots allowing various types of processor cards, I/O interface boards, and other adaptor cards to be attached.

Special adaptors known as *bridges* are available, which permit devices on the PC bus to interface to a range of more specialized industrial buses. These buses tend to be modular and rigidly specified, allowing them to be easily interfaced to industry-standard I/O devices. There are three main types of bus: STE/STD, Multibus and VME. The STE bus is an 8-bit bus capable of addressing 1 MB of memory and 4 KB of I/O space. STE was developed from the earlier 8-bit STD bus standard. Multibus also permits access to a 1 MB memory space, but allows 16-bit data transfers. Its successor, Multibus II, provides an enhanced addressing capability and is suitable for use with 32-bit processors. The VME bus has been widely used in embedded systems for some years. It is capable of 8-, 16-, 32- or 64-bit data transfers. 32-bit VME systems can achieve data transfer rates of up to 40 MB/s; 64-bit implementations can achieve twice this. Depending upon its configuration, VME can address up to 4 GB of memory, but it has no I/O space. Instead all I/O operations are memory mapped. An important variant of the VME bus is VXI. This incorporates the 32-bit VME data bus as well as a number of extensions for synchronizing and managing instruments on the bus.

Finally there are specialized implementations of PCI. Several versions of this standard bus have been developed for use in industrial embedded systems. One of the most promising of these is CompactPCI. From a functional point of view, this is very similar to a standard PCI system, although it incorporates a number of mechanical and electrical design enhancements (including a different connector, a new circuit board format and support for hot swapping of circuit boards) which make it more suited to industrial use.

It is necessary to employ a suitable interface (or bridge) in order to connect an external bus, such as Multibus or VXI, to the PC's ISA bus. The bridge performs many functions. For example, registers or buffers belonging to devices present on the external bus must be mapped into the PC's I/O space or into its memory space. Various techniques can be used. Multibus employs DMA techniques (see Chapter 6) to transfer data between the PC and the external bus. Memory mapping may be accomplished using a type of page

mapping similar to that used by the EMS. This permits regions of the external bus's memory space to be selectively mapped into a 64 KB page frame within the PC's addressable range. Alternatively, the external memory is sometimes mapped to the top of the PC's 4 GB memory space. The latter option is only possible with 80386 or later processors and with operating system software that permits 32-bit addressing. Interrupt requests on the external bus must also be mapped onto the PC's own interrupt levels (see Chapter 5 for an explanation of interrupts). Again, a number of different schemes are used. The external bus may provide more interrupt signals than are available on the PC and, in these instances, several external bus interrupts may be mapped to the same PC interrupt level. Alternatively, the external bus may support shared interrupt lines and the different interrupt allocations must be resolved by the bridge interface (possibly in conjunction with suitable software).

In general, the interface is implemented in such a way that the PC software can regard the external bus simply as an extension of its own PCI or ISA bus. Manufacturers of VME and STE bus devices may supply driver programs for use in conjunction with DOS or Windows applications running on the PC. The presence of the external bus is thus largely transparent to the DA&C programmer, although the devices connected to it (e.g. other PC boards, instruments and I/O devices) can have a profound effect on what the software is able to do. In addition, the bus implementation and bridge circuits can sometimes introduce interrupt (and other) latencies which may have to be addressed in real-time systems.

Other buses

Many other buses and communications standards, which are commonly used in PC-based DA&C systems, have not yet been mentioned: for example, IEEE-488, the Centronics parallel port, and a variety of serial buses such as RS-232, RS-422, RS-485 and USB. We will describe most of these in subsequent chapters. In addition, there are several systems and protocols, such as HART (Highway Addressable Remote Transfer) and BitBus, used in industrial sensing and control applications, as well as a number of proprietary DA&C buses (e.g. DT-Connect and Metrabus), which are outside the scope of this book.

2 Software considerations

The architecture of the PC is reasonably well suited to data acquisition. Most of the problems that occur in designing DA&C systems result from limitations imposed by software. In fact, the most serious obstacles to writing effective data-acquisition software are usually generated by the PC's operating systems. In this chapter we will discuss the main requirements of data-acquisition software and will describe some of the problems posed by using operating systems intended for desktop applications in the more demanding environment of a real-time DA&C system.

2.1 An overview of DA&C software

In addition to code that acquires data or issues control signals, it is usual for DA&C software to incorporate a number of support modules which allow the system to be configured and maintained. Other routines may be required for sorting, analysing and displaying the acquired data. A typical DA&C program may contain the following modules and facilities:

- program configuration routines
- diagnostics modules
- system maintenance and calibration modules
- run-time modules
- device drivers
- data analysis modules.

With the exception of device drivers, these modules are executed more or less independently of each other (although it is, of course, possible for multitasking systems to execute two or more concurrently). A brief overview of the main software components of a typical DA&C system is given below. Particular systems may, of

course, differ somewhat in the detail of their implementation but most applications will require at least some of these modules.

Program configuration routines

These software routines may be used for initial configuration of elements of the system that the end user would normally never (or very infrequently) have to change. This might include facilities for selecting and setting up hardware and driver options; for specifying how data is to be routed through software 'devices' (such as comparators, triggers, data-scaling operators, software latches, logical operators, or graphical displays etc.); for defining start, stop and error conditions, or for selecting delays, run times and data buffer sizes.

Diagnostic modules

Once a DA&C program has been tested and debugged, any diagnostic routines which the designer may have included for testing are often removed or disabled. However, their value should not be underestimated in 'finished' (i.e. operational) systems. Routines such as these can be invaluable tools during installation and for subsequent system maintenance. Often, the dynamic and transient nature of input/output (I/O) signals and the complex interrelation between them can make it very difficult to reproduce a fault during static testing with a voltmeter, continuity tester or a logic probe. Well-designed diagnostic routines can be a great benefit to maintenance engineers should a fault occur somewhere in the DA&C system.

With a little care and thought it is usually quite straightforward to implement a range of simple but useful diagnostic routines. These can be made to monitor aspects of the DA&C system either during normal operation or when the system is placed in a special test mode. On the simplest level, the diagnostic routines might check for incorrect hardware or software configuration. They might also be designed to perform continuous tests during normal operation of the system. This might include checking for interruptions in communication between system components, ensuring correct timing of I/O control signals, and monitoring or validating data from individual sensors.

Diagnostic software routines have their limitations, however, and other means of fault finding must be used where appropriate. Various items of test equipment such as voltmeters, logic probes, and logic pulsers may also be needed. More sophisticated equipment is sometimes required, especially when dealing with rapid pulse trains.

Digital storage, or sampling, oscilloscopes allow high frequency waveforms to be captured and displayed. These are especially suited to monitoring digital signals on high speed parallel buses or serial communications links. Where it is necessary to see the relationship between two or more time-varying signals, logic analysers may be used. These devices possess multiple (typically 32) probes, each of which detects the logic state of some element of the digital I/O circuit under test. Logic analysers are controlled by a dedicated microcomputer and can be programmed to provide a snapshot of the logic states present at the probes on a display screen. The conditions for triggering the snapshot – i.e. a selected pattern of logic states – can be programmed by the user. The device may also be used for timing analysis, in which case it operates in a similar way to a multiple-beam oscilloscope.

In addition to these items of equipment, purpose-built test harnesses may be used in conjunction with diagnostic software. Test harnesses may consist of relatively simple devices such as a bank of switches or LEDs which are used to check the continuity of digital I/O lines. At the other extreme a dedicated computer system, running specially designed test software, may be required for diagnosing problems on complex DA&C systems. See the *Software production and testing* section later in this chapter for more on this topic.

System maintenance and calibration modules

Tasks such as calibrating sensors, adjusting comparators, and tuning control loops might need to be carried out periodically by the user. Because any errors made during calibration or control loop tuning have the potential to severely disrupt the operation of the DA&C system, it is essential for the associated software routines to be as robust and simple to use as possible.

One of the most important of these system maintenance tasks is calibration of analogue input (i.e. sensor) channels. Many sensors and signal-conditioning systems need to be recalibrated periodically in order to maintain the system within its specified operating tolerance. The simplest approach (from the program designer's perspective) is to require the user to manually calculate scaling factors and other calibration parameters and then to type these directly into a data file etc. It goes without saying that this approach is both time consuming and error prone. A more satisfactory alternative is to provide an interactive calibration facility which minimizes the scope for operator errors by sampling the sensor's input at predefined reference points, and then automatically calculating the

required calibration factors. We will resume our discussion of this subject in Chapter 9 which covers scaling and interactive calibration techniques in some detail.

Run-time modules

These, together with the device drivers, form the core of any DA&C system. They are responsible for performing all of the tasks required of the system when it is 'live' – e.g. reading sensor and status inputs, executing control algorithms, outputting control signals, updating real-time displays or logging data to disk.

The nature of the run-time portion varies immensely. In some monitoring applications, the run-time routine may be very simple indeed. It might, for example, consist of an iterative polling loop that repeatedly reads data from one or more sensors and then perhaps stores the data in a disk file or displays it on the PC's screen. In many applications other tasks may also have to be carried out. These might include scaling and filtering the acquired data, or executing dynamic control algorithms.

More complex real-time control systems often have very stringent timing constraints. Many interrelated factors may need to be considered in order to ensure that the system meets its real-time response targets. It is sometimes necessary to write quite elaborate interrupt-driven buffered I/O routines or to use specially designed real-time operating systems (RTOSs) in order to allow accurate assessments of response times to be made. The software might be required to monitor several different processes in parallel. In such cases, this parallelism can often be accommodated by executing a number of separate program tasks concurrently. We will discuss concurrent programming later in this chapter.

Drivers

A diverse range of data-acquisition units and interface cards are now on the market. The basic functions performed by most devices are very similar, although they each tend to perform these functions in a different manner. The DA&C system designer may choose from the large number of analogue input cards that are now available. Many of these will, for example, allow analogue signals to be digitized and read into the PC, but they differ in the way in which their software interface (e.g. their control register and bit mapping) is implemented.

To facilitate replacement of the data-acquisition hardware it is prudent to introduce a degree of device independence into the

software by using a system of device drivers. All I/O is routed through software services provided by the driver. The driver's service routines handle the details of communicating with each item of hardware. The main program is unaware of the mechanisms involved in the communication: it only knows that it can perform I/O in a consistent manner by calling a well-defined set of driver services. In this way the data-acquisition hardware may be changed by the end user and, provided that a corresponding driver is also substituted, the DA&C program should continue to function in the same way. This provides some latitude in selecting precisely which interface cards are to be used with the software. For this reason, replaceable device drivers are commonplace in virtually all commercial DA&C programs. Protected operating systems such as Windows NT perform all I/O via a complex system of privileged device drivers.

Data analysis modules

These modules are concerned mainly with post-acquisition analysis of data. This might include, for example, spectral analysis or filtering of time-varying signals, statistical analysis (including Statistical Process Control (SPC)), and report generation. Many commercial software packages are available for carrying out these activities. Some general-purpose business programs such as spreadsheets and graphics/presentation packages may be suitable for simple calculations and for producing graphical output, but there are a number of programs which cater specifically for the needs of scientists, engineers and quality control personnel. Because of this, and the fact that the details of the techniques involved are so varied, it is impracticable to cover this subject in the present book. A variety of data reduction techniques are described by Press *et al.* (1992) and Miller (1993).

2.2 Data acquisition and control in real time

Data-acquisition systems that are designed for inspection or dimensional gauging applications may be required to gather data at only very low speeds. In these cases, the time taken to read and respond to a series of measurements may be unimportant. Because such systems usually have quite undemanding timing requirements, they tend to be relatively straightforward to implement. The choice of computing platform, operating system and programming language is usually not critical. A surprisingly large number of industrial DA&C applications fall into this category. However, many don't.

High speed DA&C normally has associated with it a variety of quite severe timing constraints. Indeed the PC and its operating systems cannot always satisfy the requirements of such applications without recourse to purpose-built hardware and/or special coding techniques. High speed processors or intelligent interface devices may be required in order to guarantee that the system will be capable of performing certain DA&C operations within specified time limits.

A real-time DA&C system is one in which the time taken to read data, process that data and then issue an appropriate response is negligible compared with the timescale over which significant changes can occur in the variables being monitored and/or controlled. There are other more precise definitions, but this conveys the essence of real-time data acquisition and control.

A typical example of a real-time application is a furnace control system. The temperature is repeatedly sampled and these readings are then used to control when power is applied to the heating element. Suppose that it is necessary to maintain the temperature within a certain range either side of some desired setting. The system detects when the temperature falls to a predefined lower limit and then switches the heating element on. The temperature then rises to a corresponding upper limit, at which point the monitoring system switches the heating element off again, allowing the temperature to fall. In this way, the temperature repeatedly cycles around the desired mean value. The monitoring system can only be said to operate in real time, if it can switch the heating element in response to changes in temperature quickly enough to maintain the temperature of the furnace within the desired operating band.

This is not a particularly demanding application – temperature changes in this situation are relatively slow, but it does illustrate the need for real-time monitoring and control systems to operate within predefined timing constraints. There are many other examples of real-time control systems in the process and manufacturing industries (such as control of reactant flow rate, controlling component assembly machines, and monitoring continuous sheet metal production, for example) which all have their own particular timing requirements. The response times required of real-time systems might vary from a few microseconds up to several minutes or longer. Whatever the absolute values of these deadlines, all real-time systems must operate to within precisely defined and specified time limits.

Requirements of real-time DA&C systems

As mentioned previously, normal PC operating systems (DOS, Microsoft Windows and OS/2) do not form an ideal basis for

real-time applications. A number of factors conspire to make the temporal response of the PC somewhat unpredictable. Fortunately there are ways in which the situation can be improved. These techniques will be introduced later in this section, but first we will consider some of the basic characteristics that a real-time computer system must possess. In addition to the usual properties required of any software, a real-time system must generally satisfy the following requirements.

Requirement 1: high speed

The most obvious requirement of a real-time system is that it should be able to provide adequate throughput rates and response times. Fortunately, many industrial applications need to acquire data at only relatively low speeds (less than one or two hundred readings per second) and need response times upwards of several tens of milliseconds. This type of application can be easily accommodated on the PC. Difficulties may arise when more rapid data acquisition or shorter responses are required.

Obviously a fast and efficient processor is the key to meeting this requirement. As we have already seen, modern PCs are equipped with very powerful processors which are more than adequate for many DA&C tasks. However, the memory and I/O systems, as well as other PC subsystems, must also be capable of operating at high speed. The disk and video subsystems are notorious bottlenecks, and these can severely limit data throughput when large quantities of data are to be displayed or stored in real time. Fortunately, most modern PC designs lessen this problem to some extent by making use of high speed buses such as the Small Computer Systems Interface (SCSI) and the PCI local bus. Modern Pentium-based PCs are very powerful machines and are capable of acquiring and processing data at ever increasing rates. Older XT and 80286- or 80386-based computers offer a lower level of performance, but are still often adequate in less demanding applications.

Requirement 2: determinism

A deterministic system is one in which it is possible to precisely predict every detail of the way in which the system responds to specific events or conditions. There is an inherent predictability to the sequence of events occurring within most computer programs, although the timing of those events may be more difficult to ascertain. A more practical definition of a deterministic system is one in which the times taken to respond to interrupts, perform task switches and execute operating system services etc. are well known and guaranteed. In

short, a deterministic system has the ability to respond to external events within a *guaranteed* time interval.

Determinism is an important requirement of all real-time systems. It is necessary for the programmer to possess a detailed knowledge of the temporal characteristics of the operating system and device drivers as well as of the DA&C program itself. This knowledge is an important prerequisite for the programmer to assess the worst-case response of the system and thus to ensure that it meets specified deadlines.

Requirement 3: high resolution timekeeping and pacing facilities

In addition to being able to operate within given time constraints, it is important for most real-time systems to be able to precisely *measure* elapsed time. This ability is essential for the software to accurately schedule I/O operations and other tasks. Where data is acquired at irregular or unpredictable rates, it is particularly important to be able to time stamp readings and other events. An accurate timing facility is also an invaluable aid to fault finding in dynamic systems. The PC is equipped with a real-time clock and a set of timers which are useful for this purpose. The timers function by means of the PCs interrupt system and provide a powerful means of pacing a data-acquisition sequence or for generating precisely timed control signals. The PC's timing facilities are discussed briefly in Chapter 3.

Requirement 4: flexible interfacing capability

It should be obvious that any data-acquisition and control system should be able to interface easily to sensors, actuators and other equipment. This requirement covers not only the PC's physical interfacing capacity (i.e. the presence of appropriate plugs, sockets and expansion slots), but also encompasses an efficient means of transferring data in and out of the computer.

The PC possesses a very flexible interfacing system. As mentioned previously, this is implemented by means of the standard ISA, EISA, MCA or PCI expansion buses or PCMCIA slots. The PC also facilitates processor-independent high speed I/O using techniques known as Direct Memory Access (DMA) and bus mastering. These facilities give the PC the capability to interface to a range of external buses and peripherals (e.g. data-logging units, sensors, relays and timers) via suitable adaptor cards. Indeed, adaptor cards for RS-232 ports and Centronics parallel ports, which can be used to interface to certain types of DA&C hardware, are an integral component of almost all PCs. Interfacing, data transfer and DMA are discussed in more detail in Chapters 3, 6, 7 and 8.

Requirement 5: ability to model real-world processes

It should also be apparent to the reader that the logical structure of a real-time DA&C system should adequately mirror the processes that are being monitored. As we shall see on the following pages this requirement sometimes necessitates using a specially designed real-time operating system. In less demanding applications, however, such a step is unnecessary provided that due care is taken to avoid some of the pitfalls associated with standard 'desktop' operating systems.

Requirement 6: robustness and reliability

Again, this is a rather obvious requirement but its importance cannot be overstated. A number of steps can be taken to maximize the reliability of both hardware and software. We will return to this issue later in this chapter.

Simple DA&C systems

Some PC-based DA&C systems are fairly undemanding in regard to the detailed timing of I/O events. Many applications involve quite low speed data logging, where samples and other events occur at intervals of several seconds or longer. In other cases a high *average* data-acquisition rate might be needed, but the times at which individual readings are obtained may not be subject to very tight restrictions. Often, only a single process (or a group of closely coupled processes) will have to be monitored and in these cases it is usually sufficient to base the run-time portion of a DA&C program on a simple polling loop as illustrated in Figure 2.1.

This figure shows the sequence and repetitive nature of events that might occur in a simple single-task application. When some predefined start condition occurs (such as a keystroke or external signal) the program enters a monitoring loop, during which data is acquired, processed and stored. The loop may also include actions such as generating signals to control external apparatus. The program exits from the loop when some desired condition is satisfied – i.e. after a certain time has elapsed, after a predefined number of readings have been obtained or when the user presses a key. In some cases, additional processing may be performed once the data-acquisition sequence has terminated.

There are, of course, many variations on this basic theme, but the essence of this type of program structure is that all processing is performed within a single execution thread. This means that each instruction in the program is executed in a predefined sequence,

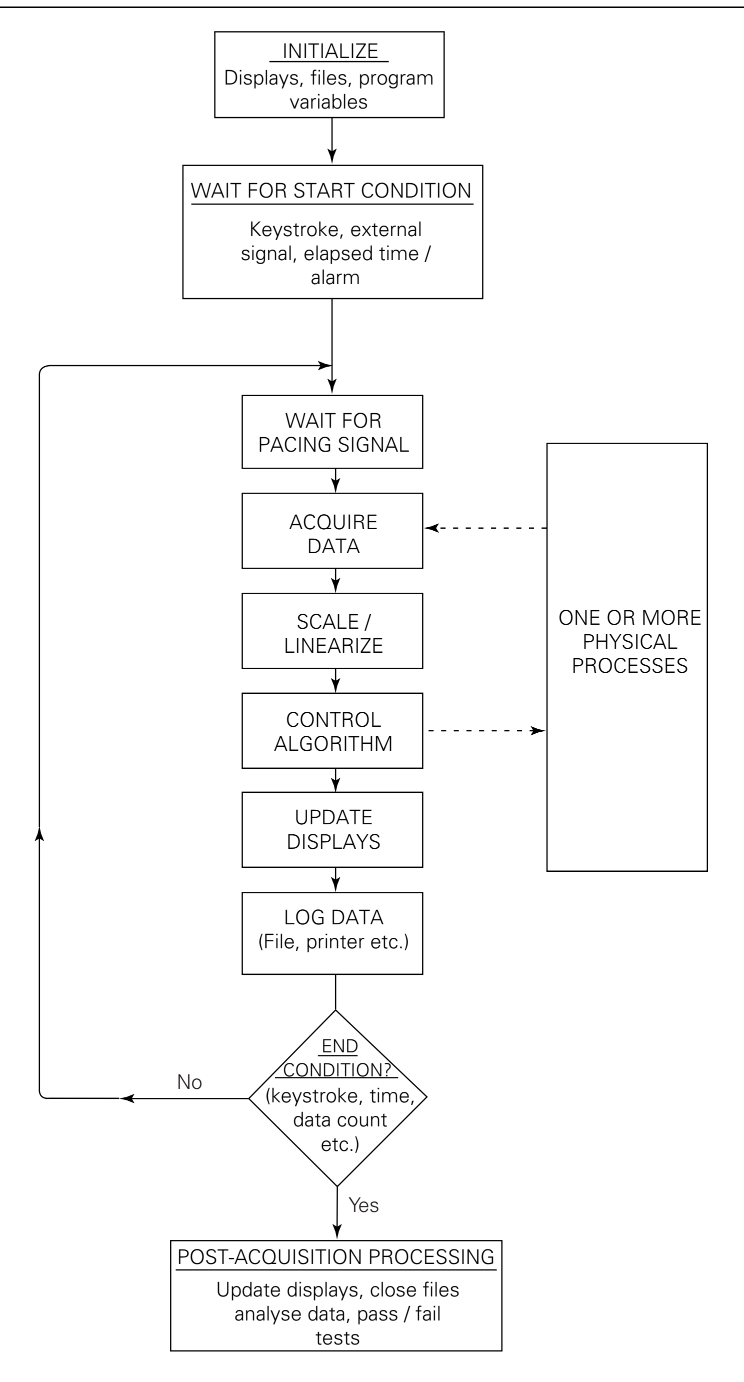


Figure 2.1 *Schematic illustration of the structure of a typical DA&C program based on a simple polling loop*

one after the other. There is no possibility that external events will cause parts of the program to be executed out of sequence. Any tasks which the computer does carry out in parallel with the execution of the program, such as responding to keystrokes, 'ticks' of the in-built timer or to other system interrupts, are essentially part of the operating system and are not *directly* related to the functioning of the DA&C program.

It should be noted that events such as a timer or keyboard interrupt will temporarily suspend execution of the DA&C program while the processor services the event (increments the time counter or reads the keyboard scan code). This means that the timing of events within the interrupted program will not be totally predictable. However, such a system is still considered to operate in real time if the uncertainty in the timing of the data-acquisition cycle is small compared with the timescales over which the monitored variables change.

Systems with more stringent timing requirements

All real-time systems have precisely defined timing requirements. In many cases, these requirements are such that the system must be designed to respond rapidly to events which occur asynchronously with the operation of the program. In these cases, a simple polling loop may not guarantee a sufficiently short response time. The usual way to achieve a consistent and timely response is to use interrupts.

Interrupts

Interrupts are the means by which the system timer, the keyboard and other PC peripherals request the processor's attention. When service is required, the peripheral generates an interrupt request signal on one of the expansion bus lines. The processor responds, as soon as possible, by temporarily suspending execution of the current program and then jumping to a predefined software routine. The routine performs whatever action is necessary to fulfil the request and then returns control to the original program, which resumes execution from the point at which it was interrupted.

Because an interrupt handling routine is executed in preference to the main portion of the program, it is considered to have a higher priority than the non-interrupt code. The PC has the capacity to deal with up to 15 external interrupts (8 on the IBM PC, XT and compatibles) and each of these is allocated a unique priority. This prioritization scheme allows high priority interrupts to be allotted to the most time-critical tasks. With appropriate software techniques,

the programmer may adapt and modify the interrupt priority rules for use in real-time applications.

The PC is equipped with a very flexible interrupt system, although the gradual evolution of the PC design has left something to be desired in terms of the allocation of interrupts between the processor and the various PC subsystems. When using interrupts, you should bear in mind two important considerations (although there are many others): re-entrancy and interrupt latency. These topics are introduced below. The PC's interrupt system, and the problems of re-entrancy and interrupt latency, are described in more detail in Chapter 5.

Re-entrant code and shared resources

This is relevant to all types of software, not just to real-time DA&C programs. Because external interrupts occur asynchronously with the execution of the program, the state of the computer is undefined at the time of the interrupt. The interrupt handling routine must, therefore, ensure that it does not inadvertently alter the state of the machine or any software running on it. This means that it must (a) preserve all processor registers (and other context information), and (b) refrain from interfering with any hardware devices or data to which it should not have access. The last requirement means that care should be taken when calling any subroutines or operating system services from within the interrupt handler. If one of these routines happened to be executing at the time that the interrupt occurred, and the routine is then re-entered from within the interrupt handler, the second invocation may corrupt any internal data structures that the routine was originally using. This can obviously cause severe problems – most likely a system crash – when control returns to the interrupted process. Of course, software routines *can* be written to allow multiple calls to be made in this way. Such routines are termed re-entrant.

Unfortunately most MS-DOS and PC-DOS services are not re-entrant, and so calls to the operating system should generally be avoided from within interrupt handlers. Specially designed real-time operating systems (RTOSs) are available for the PC and these normally incorporate at least partially re-entrant code. The run-time libraries supplied with compilers and other programming tool kits may not be re-entrant. You should always attempt to identify any non-re-entrant library functions that you use and take appropriate precautions to avoid the problems outlined above. A similar consideration applies when accessing any system resource (including hardware registers or operating system or BIOS data) which may be used by the main program and/or by one or more interrupt handlers.

Interrupt latencies

This consideration is more problematic in real-time systems. The processor may not always respond immediately to an interrupt request. The maximum time delay between assertion of an interrupt request signal and subsequent entry to the interrupt handler routine is known as the interrupt latency. The length of the delay depends upon the type of instructions being executed when the interrupt occurs, the priority of the interrupt relative to the code currently being executed, and whether or not interrupts are currently disabled. Because interrupts are asynchronous processes, the effect of these factors will vary. Consequently, the delay in responding to an interrupt request will also vary. In order to ensure that the system is able to meet specified real-time deadlines, it is important for the system designer to quantify the *maximum* possible delay or interrupt latency.

By careful design it is possible to ensure that the code within a DA&C program does not introduce excessive delays in responding to interrupts. However, most programs occasionally need to call operating system or BIOS services. The programmer must ensure that the system will still respond within a specified time, even if an interrupt occurs while the processor is executing an operating system service. Unfortunately, standard desktop operating systems such as DOS and Microsoft Windows are not designed specifically for real-time use. These operating systems generally exhibit quite long interrupt latencies (particularly Windows). Typical figures are in the order of 10–20 ms, although you should not place too much reliance on this value as it will vary quite considerably between applications. Unfortunately, interrupt latency data for Windows and MS-DOS is hard to come by. Such operating systems are known as non-deterministic.

The magnitude of the problem can be reduced if real-time operating systems (RTOSs) are used. These operating systems are designed so as to minimize interrupt latencies. They are usually essential if latencies of less than about 1 ms are required. The interrupt latencies applicable to various parts of the RTOS are also generally documented in the operating system manual, allowing the programmer to ensure that the whole system is capable of meeting the required response deadlines.

Concurrent processing

Systems monitored or controlled by real-time DA&C software often consist of a number of separate processes operating in parallel. If these processes are asynchronous and largely independent of each

other it may be very difficult to represent them adequately in a simple, single-threaded program. It is usually more convenient to model parallel processes within the computer as entirely separate programs or execution threads. This arrangement is illustrated in Figure 2.2 which shows three separate processes being executed in parallel (i.e. three separate instances of the single-task loop of Figure 2.1).

Ideally, each process would be executed independently by a separate computer. We can go some way towards this ideal situation by delegating specific real-time tasks to distributed intelligent data-logging or control modules. Many factory automation systems adopt this approach. Dedicated data-acquisition cards, with on-board memory buffers and an intrinsic processing ability, can also be used to provide a degree of autonomous parallel processing. Other parallel processing solutions are also available, but these generally involve the use of separate multiprocessing computer systems and, as such, are beyond the scope of this book.

The most common way of modelling parallel processes on the PC is to employ concurrent programming (or multitasking) techniques. Most modern PCs are equipped with 80386, 80486 or Pentium processors and these incorporate features which greatly facilitate multitasking. On single-processor systems such as the PC, concurrent execution is achieved by dividing the processor's time between all executing programs. The processor executes sections of each program (or *task*) in turn, switching between tasks frequently enough to give the impression that all tasks are being executed simultaneously. This technique is used in multitasking operating systems such as OS/2, Windows and UNIX.

Scheduling

Clearly, there must be a set of rules governing how and when task switching is to occur. These rules must also define the proportions of time assigned to, and the priorities of, each program. The process of allocating execution time to the various tasks is known as scheduling and is generally the responsibility of the operating system. The basic principles of scheduling are quite straightforward although the details of its implementation are somewhat more complex.

There are several ways in which a task scheduler can operate. In a system with pre-emptive scheduling, the operating system might switch between tasks (almost) independently of the state of each task. In non-pre-emptive scheduling, the operating system will perform a task switch only when it detects that the current task has reached a suitable point. If, for example, the current task makes a call to an operating system service routine, this allows the operating system to

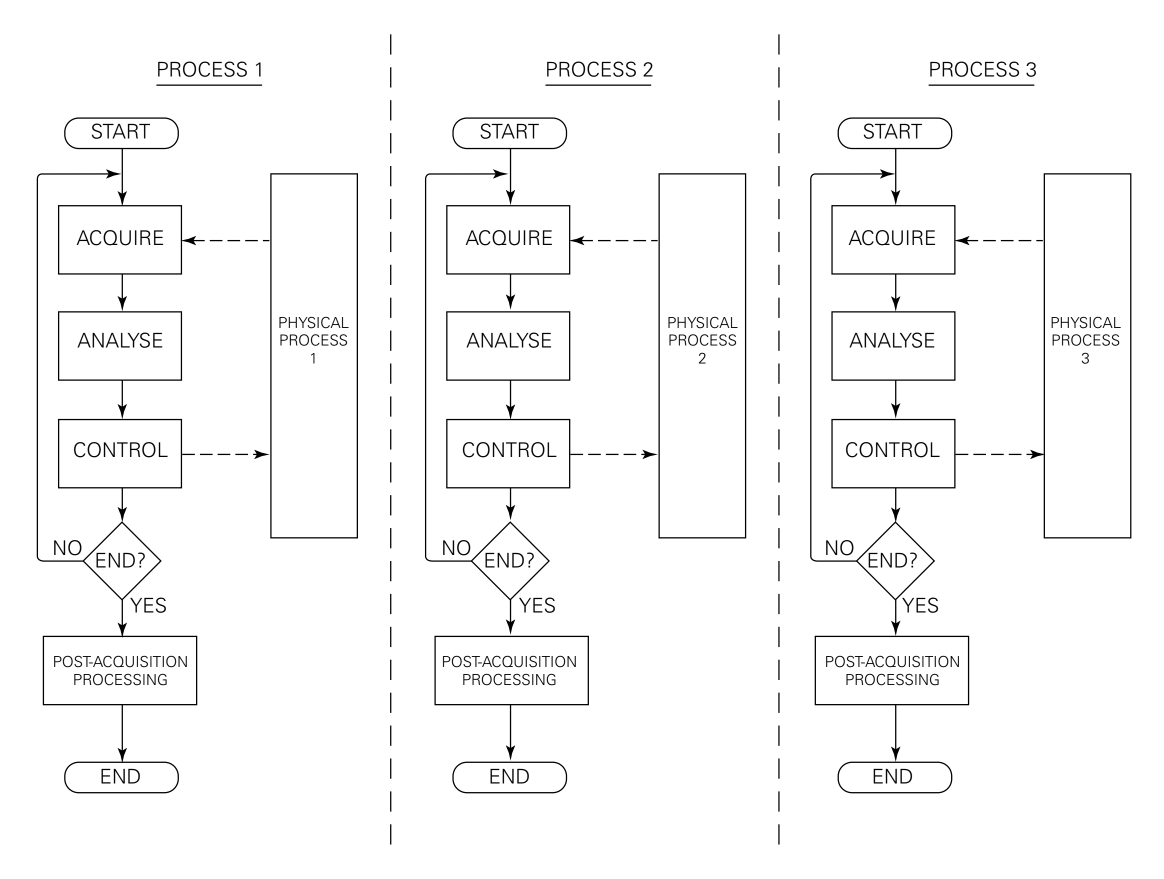


Figure 2.2 *Schematic illustration of concurrent monitoring and control of parallel processes*

check whether the task is idle (e.g. waiting for input). If it is idle, the operating system may then decide to perform more useful work by allowing another process to execute. This makes for efficient use of available processor time, but, as it relies on an individual task to initiate the switch, it does allow poorly behaved tasks to hog the processor. This is obviously undesirable in real-time applications because it may prevent other processes from executing in a timely manner. Pre-emptive scheduling, on the other hand, provides for a fairer division of time between all pending processes, by making the operating system responsible for regularly initiating each task switch.

Task switching, threads and processes

Whenever the operating system switches between tasks it has to save the current context of the system (including processor registers, pointers to data structures and the stack), determine which task to execute next, and then reload the previously stored context information for the new task. This processing takes time, which in a real-time operating system should be as short as possible. Most multitasking 'desktop' operating systems use the advanced multitasking features available on 80386 and later processors to implement a high degree of task protection and robust task switching. However, this type of task switching can be too time consuming for use in high performance real-time systems.

Other operating systems, such as those designed for real-time use, minimize the switching overhead by allowing each process (i.e. executing program) to be divided into separate execution *threads*. Threads are independent execution paths through a process. They can generally share the same code and data areas (although they each tend to have their own stack segment), and are normally less isolated from each other than are individual processes in a multitasking system. There is also less context information to be saved and restored whenever the operating system switches between different threads, rather than between different processes. This reduces the amount of time taken to perform the context switch. Although not intended for hard real-time applications, Microsoft Windows NT supports multi-threaded processes.

The term 'task' is used somewhat loosely in the remainder of this chapter to refer to both processes and threads.

Real-time design considerations: a brief overview

As mentioned previously many PC-based data-acquisition systems will *not* be required to operate within the very tight timing constraints imposed in real-time control applications. However, it is useful for

programmers involved in producing any type of time-dependent application to have a basic understanding of the fundamentals of real-time design. Even if you do not plan to implement these principles in your own systems, the following introduction to the subject may help you to avoid any related potential problems.

Structure of real-time multitasking programs

A typical real-time system might consist of several tasks running in parallel. The division of processing between tasks will usually be assigned on the basis of the real-world processes which the system must model. Each task will often be assigned to a separate, and more or less independent, *physical* process.

A typical example is the control of a manufacturing process for producing rolled metal or polymer sheet. One task might be dedicated to monitoring and controlling product thickness. Another may be assigned to regulating the temperature to which the material is heated prior to being passed through the rollers. Yet another task could be used for periodically transferring thickness, temperature and status information to the display. A similar arrangement is shown in Figure 2.3.

The interface between the various tasks and the data-acquisition hardware is often implemented by means of one or more interrupt handlers. These are normally contained within some form of dedicated device driver and are designed to allow the system to respond quickly to external events. Data acquired via an interrupt handler might be stored in a memory buffer until the associated task is able to read and process it. The individual tasks are responsible for operations such as data logging, display maintenance or data reduction. A task might also be assigned to perform real-time calculations or tests on the acquired data. The results can then be used as the basis for generating control signals which are output to external equipment. In general, time-critical operations are performed by high priority tasks, allowing them to take precedence over less critical operations such as managing the user interface.

There is generally a need for some form of intertask communication. This facility is often based on the use of message queues and memory buffers. Where shared memory or other resources are used, special protection mechanisms must be employed to mediate between tasks. Interprocess communication and protection mechanisms are provided by real-time operating systems (RTOSs). We will consider some of these facilities in more detail in the following sections. Additional information on real-time and multitasking systems can be found in the texts by Evesham (1990), Adamson (1990), Ben-Ari (1982) and Bell *et al.* (1992).

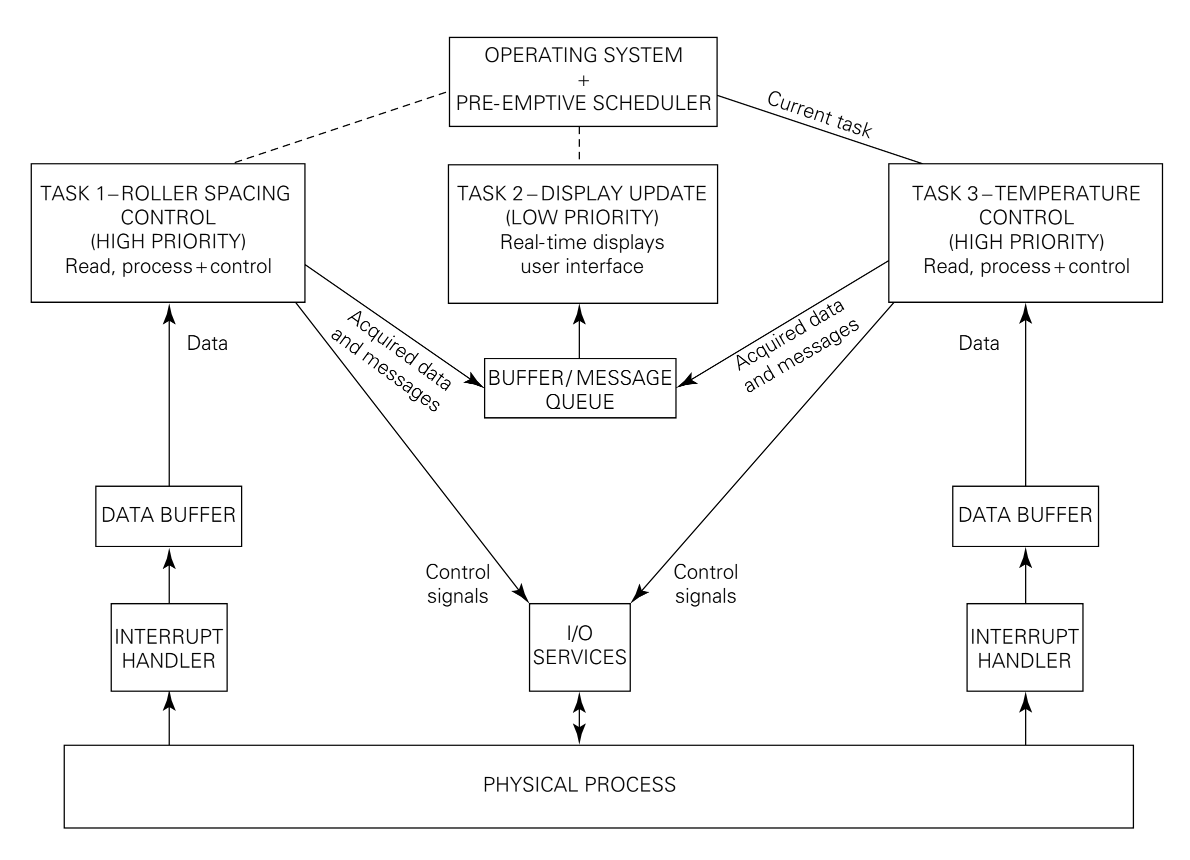


Figure 2.3 *Conceptual structure of a typical real-time multitasking system*

Accessing shared resources and interprocess communication

Although the processes in a multitasking system tend to operate more or less independently of each other, there usually has to be some degree of communication between them in order to transfer data or to synchronize certain features of their operation.

Interprocess communication involves accessing a shared resource such as a buffer or message queue that is maintained somewhere in the PC's memory. The operating system is generally responsible for coordinating access to these structures, and to other system resources such as disk drives etc.

Whenever a task or an interrupt handler needs to access any shared resource – including hardware, operating system services and data structures – great care must be taken to avoid conflicting with any other tasks that may be in the process of accessing the same resource. Consider a section of code that accesses a shared resource. If the code could possibly malfunction as a result of being pre-empted (or interrupted) by a task that accesses the same resource, the code is known as a critical section. It is necessary to protect critical sections from this type of interference by temporarily blocking task switches and/or interrupts until the critical section has been completed. This requirement is known as mutual exclusion.

Mutual exclusion can be enforced by means of semaphores. These are essentially flags or tokens that are allocated by the operating system to any process wishing to access a particular resource. A task may not proceed into a critical section until it has obtained the appropriate semaphore. In some systems, implementations of semaphores, for the purpose of enforcing mutual exclusion, are referred to as Mutexes.

Deadlocks and lockouts

A deadlock occurs when all processes within a system become suspended as a result of each process waiting for another to perform some action. A lockout is similar, but does not affect all tasks. It arises when conditions brought about by two or more processes conspire to prevent another process from running. Great care must be taken to avoid the possibility of deadlocks or lockouts in any real-time system.

Priorities

Many multitasking systems allow priorities to be assigned to the individual tasks. Whenever the scheduler performs a task switch it uses the priorities assigned to each task to decide which one to execute next. This has the obvious benefit in real-time systems of allowing the most important or time-critical tasks to take precedence.

In some systems, priorities can be changed dynamically. Priority systems can be quite complex to implement and a number of special programming techniques may have to be used, both within the application program and within the operating system itself, to ensure that the priorities are always applied correctly.

A common problem is priority inversion. If a low priority task holds a semaphore and is then pre-empted by a higher priority task that requires the same semaphore, the operating system will have to let the low priority task continue to run until it has released the semaphore. If, meanwhile, the low priority task is pre-empted by a task with an intermediate priority, this will run in preference to the highest priority task. Some of the solutions to priority inversion (such as priority inheritance which dynamically alters the priority of tasks) raise additional problems. Certain RTOSs go to great lengths to provide generally applicable solutions to these problems. However, many of these difficulties can be avoided if the programmer has a detailed understanding of all of the software components running on the system so that potential deadlocks or other incompatibilities can be identified.

2.3 Implementing real-time systems on the PC

Thanks to its expansion bus and flexible interrupt system, the PC has a very open architecture. This allows both hardware and software subsystems to be modified and replaced with ease. Although this openness is a great benefit to designers of DA&C systems, it can introduce problems in maintaining the system's real-time performance. If non-real-time code is introduced into the system, in the form of software drivers which trap interrupts or calls to operating system services, it may no longer be possible to guarantee that the system will meet its specified real-time targets. It should be clear that there is a need to exercise a considerable degree of control over the software subsystems that are installed into the PC.

In general, the architecture of the PC itself is reasonably well suited to real-time use. Its operating system is often the limiting factor in determining whether the PC can meet the demands of specific real-time applications. Standard MS-DOS or PC-DOS, Microsoft Windows and the PC's BIOS present a number of difficulties which may preclude their use in some real-time systems. However, there are several specially designed real-time operating systems (RTOSs), including real-time versions of DOS and the BIOS, which can help to alleviate these problems. Real-time operating systems can be quite complex, and different implementations vary to such a degree that

it is impracticable to attempt a detailed coverage here. The reader is referred to manufacturer's literature and product manuals for details of individual RTOSs.

As we have already noted, standard desktop operating systems (e.g. MS-DOS and Microsoft Windows) were not designed specifically for real-time use. Interrupt latencies and re-entrancy can be problematic. These operating systems frequently embark on lengthy tasks, which can block interrupt processing for unacceptable (and possibly indeterminate) lengths of time. Some of the instructions present on 80386 and subsequent processors, which were designed to facilitate multitasking (and which are used on systems such as Windows, OS/2 and UNIX), are not interruptible and can occupy several hundred processor cycles. Using these operating systems and instructions can increase interrupt latencies to typically several hundred microseconds or more.

Table 2.1 lists a few example applications which require different degrees of timing precision and different sampling rates. Notice that where timing constraints are more relaxed, non-deterministic operating systems such as Windows may be used in conjunction with slow software-controlled DA&C hardware. Tighter timing constraints (near the bottom of the table) necessitate the use of buffered DA&C cards, hardware triggering, autonomous data loggers or specialized RTOSs. Note that the timing figures and sampling rates listed in the table are intended only as a rough guide and in reality may vary considerably between applications.

The BIOS

The PC's BIOS can be a source of problems in real-time applications. Several of the BIOS services can suspend interrupts for unpredictable lengths of time. Some of the BIOS may also be non-re-entrant. At least one manufacturer produces a real-time version of the BIOS for use with its real-time DOS, and another supplies an independent real-time BIOS that can be used with MS-DOS or compatible systems (including real-time DOSes). These BIOSes provide many standard low level I/O facilities while maintaining a short and guaranteed interrupt latency.

DOS

MS-DOS is a relatively simple operating system designed for execution in real mode. It is largely non-re-entrant, and it does not possess multitasking capabilities or the deterministic qualities (e.g. a short and well-defined interrupt latency) required for real-time use.

Table 2.1 *DA&C applications representative of various timing regimes*

Application	*Approx. sampling rate (samples s^{-1})*	*Permissible timing uncertainty*[1] *(ms)*	*Possible operating system and hardware combination*
Static dimensional gauging	Not applicable	Few × 1000	MS-DOS, Windows 98 or Windows NT. Low speed (non-buffered) ADC card or multichannel serial port data logger.
Furnace temperature control	<1	100	MS-DOS, Windows 98 or Windows NT. Low speed (non-buffered) ADC card or RS-485 intelligent temperature sensing module.
Low speed chemical process control	1–5	50	MS-DOS with low speed non-buffered ADC card or serial port data acquisition/control modules. Windows NT with buffered and hardware-triggered DA&C card or autonomous data logger/controller.
Roller control in sheet metal production	5–50	2–10	MS-DOS with medium speed software-triggered DA&C card and SSH, or RS-232 data logger. Windows NT with hardware-triggered buffered DA&C card and SSH, or autonomous data logger/controller.
Load monitoring during manual component testing	10–50	2–5	MS-DOS or Windows NT/98 with hardware-triggered, buffered DA&C card or IEEE-488 instrumentation.
Dynamic load/displacement monitoring with machine control	10–200	1–2	MS-DOS or Windows NT/98 with hardware-controlled, buffered DA&C card.
Destructive proof testing and machine control	>1000	<1	RTOS with high speed, hardware-triggered and buffered ADC card and opto-isolated I/O cards.
Audio testing (no control)	>1000	<1	MS-DOS, Windows NT or RTOS with fast, buffered ADC card.

[1] Of a single measurement, assuming accurate *average* sampling rate is maintained.

Nevertheless, it is inexpensive and is often suitable as the basis for simple DA&C systems provided that the real-time requirements are not too stringent. For many low and medium speed data-acquisition applications, in which timing accuracies of the order of 10 ms or so are needed, DOS is ideal, being both relatively simple and compact. Real-time *control* applications are often more demanding, however.

If timing is critical, it may be prudent to turn to one of the specially designed real-time versions of DOS. These tend to be ROMable and suitable for use in embedded PC systems. It should be noted, though, that not all ROMable DOSes are fully deterministic – i.e. interrupt latencies and other timing details may not be guaranteed.

There are now several real-time versions of DOS on the market such as General Software Inc.'s Embedded DOS and Datalight Inc.'s ROM-DOS (available in the UK from Great Western Instruments Ltd and Dexdyne Ltd, respectively). Real-time DOS systems are fully deterministic, having well-defined interrupt latencies, and are generally characterized by their ability to execute multiple processes using pre-emptive task scheduling. Other facilities, such as task prioritization and the option to utilize non-pre-emptive scheduling are also often included.

The multitasking capabilities of real-time DOSes contrasts with those of desktop operating systems. Because the requirements of most real-time applications are relatively simple, the large quantities of memory and the task protection features offered by heavy-weight operating systems like Windows and OS/2 can often be dispensed with.

Real-time DOSes are designed to minimize task switching overheads. Each task switch may be accomplished in a few microseconds and interrupt latencies are often reduced to less than about 20 μs, depending, of course, on the type of PC used. Detailed timing information should be provided in the operating system documentation.

These operating systems are also generally re-entrant to some extent. This allows DOS services to be shared between different tasks and to be safely called from within interrupt handlers. Other features found in real-time DOSes may include mutual exclusion primitives (semaphores) for accessing shared resources and for protecting critical sections; software timers; interprocess communication features such as support for message queues; and debugging facilities. These operating systems also support a range of other configurable features which allow the operating system to be adapted for use in a variety of different real-time or embedded systems.

Real-time DOSes retain a high degree of compatibility with MS-DOS's interrupts, file system and installable device drivers. Networks

may also be supported. Note that version numbers of real-time DOSes may bear no relation to the version of MS-DOS which they emulate. Some systems provide basic MS-DOS version 3.3 compatibility while others also provide some of the features found in more recent releases of MS-DOS.

In some cases, at least partial source code may also be available, allowing the operating system itself to be adapted for more specialized applications. The main drawback with real-time versions of DOS is that they can be considerably more expensive, particularly for use in one-off systems. Royalties may also be payable on each copy of the operating system distributed.

DOS extenders and DPMI

With the proliferation of sophisticated multitasking operating systems, DOS extenders are now used much less frequently than they were in the early 1990s. However, if you have to develop a DOS-based DA&C system, an extender will allow you to access up to typically 16 MB of memory. This is achieved by running your program in protected mode and, when necessary, switching back to real mode in order to access DOS and BIOS services. DOS extenders conforming to the DOS Protected Mode Interface (DPMI) standard are available from several vendors.

In spite of having a slightly greater potential for determinism than processes running under Windows, for example, a DPMI-based program may run more slowly that its real-mode counterpart. A number of the problems outlined for Windows in the following section also apply to DOS extenders. Mode switches are required whenever DOS or BIOS services are called, or when the system has to respond to interrupts. Some DOS extenders may also virtualize the interrupt system, by providing services specifically for disabling and enabling interrupts. To this end, they also prevent the program from directly disabling or enabling interrupts by trapping the `STI` and `CLI` instructions in much the same way as the processor might trap `IN` and `OUT` instructions in protected mode. This point should be borne in mind as it can affect the system's interrupt performance. DOS extenders are discussed in detail in the text by Duncan *et al.* (1990).

Microsoft Windows

Microsoft Windows 98 and Windows NT version 4 are the latest releases in a long line of graphical windowing environments for the PC. Since it was first introduced in 1985, Windows has evolved from a simple shell sitting on top of DOS into a very powerful and

complex operating system. The oldest version of Windows that is still used in significant numbers is Windows 3.1. This version, which was released in 1992, introduced many of the features present in Windows today such as TrueType fonts and Object Linking and Embedding (OLE). Windows for Workgroups, was subsequently released in 1992. This included support for peer-to-peer networking, fax systems and printer sharing, but in most other respects was similar to Windows 3.1.

Subsequently, Windows development split, forming two product lines, Windows 9x and Windows NT. At the time of writing the latest releases are Windows 98 (which supersedes Windows 95) and Windows NT version 4 (version 5 is due for imminent release). Although Windows 98 and NT are distinctly different products they share many similarities. Both are 32-bit protected mode operating systems, supporting a 4 GB flat memory model, sophisticated security features and support for installable file systems and long (256 character) file names. Both also use the same applications programming interface: the Win32 API.

Several features of Windows NT and Windows 98 are important in the context of real-time data acquisition and control. The ability to pre-emptively multitask many threads and to interface to a range of peripherals in a device-independent manner are especially relevant. However, there are a number of quite serious problems associated with using any of the current versions of Windows in real time. Rather than having complete control of the whole PC (as is the case with real-mode DOS programs, for example), programs running under Windows execute under the control and supervision of the operating system. They have restricted access to memory, I/O ports and the interrupt subsystem. Furthermore, they must execute concurrently with other processes and this can severely complicate the design of DA&C programs. In order to build a deterministic Windows system, it is necessary to employ quite sophisticated programming techniques. The following sections outline some of the problems associated with using Windows in real time.

While Windows NT and 98 are both essentially desktop operating systems, Windows NT is the more robust of the two and is widely regarded as a well engineered, secure and reliable operating system. It contains pure 32-bit code, and possesses integrated networking capabilities and enhanced security features. Windows NT has also been designed to be portable across platforms, including multiprocessor and RISC systems. For these reasons Windows NT is often used in preference to Windows 98 for industrial interfacing applications.

A brief introduction to data acquisition under Windows is provided in the following subsections. Those readers interested in programming under Windows are advised to consult one of the numerous books on this topic such as Solomon (1998), Templeman (1998), Petzold (1996) or Oney (1996).

Windows overview

One of the main features of Windows NT and Windows 98 is their ability to run 32-bit software. This offers significant (potential) improvements in execution speed as well as many other advantages.

In contrast to Windows 95/98, Windows NT contains only 32-bit code. This is beneficial since 16-bit portions of code within Windows 95/98 can have an adverse effect on performance. Problems can arise when 32-bit code has to communicate with 16-bit code, and vice versa. The process which permits such a communication is known as a *thunk*. This is a complex action which, as it involves switching between 16-bit and 32-bit addressing schemes, can slow program execution considerably. In fact, it has been reported that Windows 95 can multitask 16-bit applications as much as 55 per cent slower than they would run under Windows 3.1.

32-bit code offers many advantages to the programmer. Foremost among these is the ability to use a flat memory addressing scheme. This gives access to up to 4 GB of memory without the need to continually reload segment registers. Access to memory is closely supervised and controlled at the page level by the operating system. Page level protection is implemented using the processor's page translation and privilege ring mechanisms. These actually virtualize the memory map so that the memory addresses used by application programs do not necessarily correspond to physical memory addresses. All memory accesses are performed indirectly by reference to a set of page tables and page directories that are maintained by the operating system. Under this scheme it is impossible for an application to access (and thereby corrupt) memory belonging to another 32-bit application. Memory management under Windows is a complex business, but fortunately much of the mechanism is hidden from the programmer.

Virtualization is not confined to memory. Windows 98 and NT use features of the 80486 and subsequent processors to virtualize the PC's I/O and interrupt subsystems. All of this virtualization allows the operating system to completely isolate application programs from the hardware. A complete virtual machine is created in which to run each application. Although virtualization is efficient and makes for a robust environment for multitasking, it does introduce additional

overheads, and these can be difficult to overcome in real-time data acquisition.

As we have seen in Chapter 1, the 80486 and Pentium processors provide several mechanisms that facilitate multitasking and task protection. Among these are the assignment of privilege levels to different processes. The privilege level scheme allows operating system processes to take precedence over the less privileged application program. There are four privilege levels known as Rings 0, 1, 2 and 3. Windows uses only two of these: Ring 0 (also termed Kernel Mode under Windows NT) for highly privileged operating system routines and drivers; and Ring 3 (also termed User Mode) for applications programs and some operating system code. This is illustrated in Figure 2.4. Compare the Windows NT and 98 architecture with that of a real-mode DOS system. In the latter case, the application effectively runs at the same privilege level as the operating system, and it can access any part of the PC's hardware, BIOS or operating system without restriction.

Multitasking and scheduling

Windows 3.1 utilizes a non-pre-emptive scheduling mechanism. The method employed is essentially cooperative multitasking in which the currently active task has the option to either initiate or block further task switches. Because of this, it is possible for an important DA&C task to be blocked while some less time-critical task, such as rearranging the user interface, is carried out. Under this scheme it is, therefore, difficult to ensure that data is acquired, and that control signals are issued, at predictable times.

Windows NT and Windows 98, however, employ a greatly improved multitasking scheduler. 32-bit applications are multitasked pre-emptively, which yields greater consistency in the time slicing of different processes. The pre-emptive scheduler implements an idle detection facility, which diverts processor time away from tasks that are merely waiting for input. Another benefit is the ability to run multiple threads within one application. It is important to bear in mind that pre-emptive multitasking applies only to 32-bit programs. The older style 16-bit programs are still multitasked in a non-pre-emptive fashion and cannot incorporate multiple threads.

Windows NT and Windows 98 also employ more robust methods of interprocess communication. Windows 3.1 supported a system of messages that were passed between processes in order to inform them of particular events. As these messages were stored in a single queue, it left the system vulnerable to programs that did not participate efficiently in the message passing protocol. Windows NT and

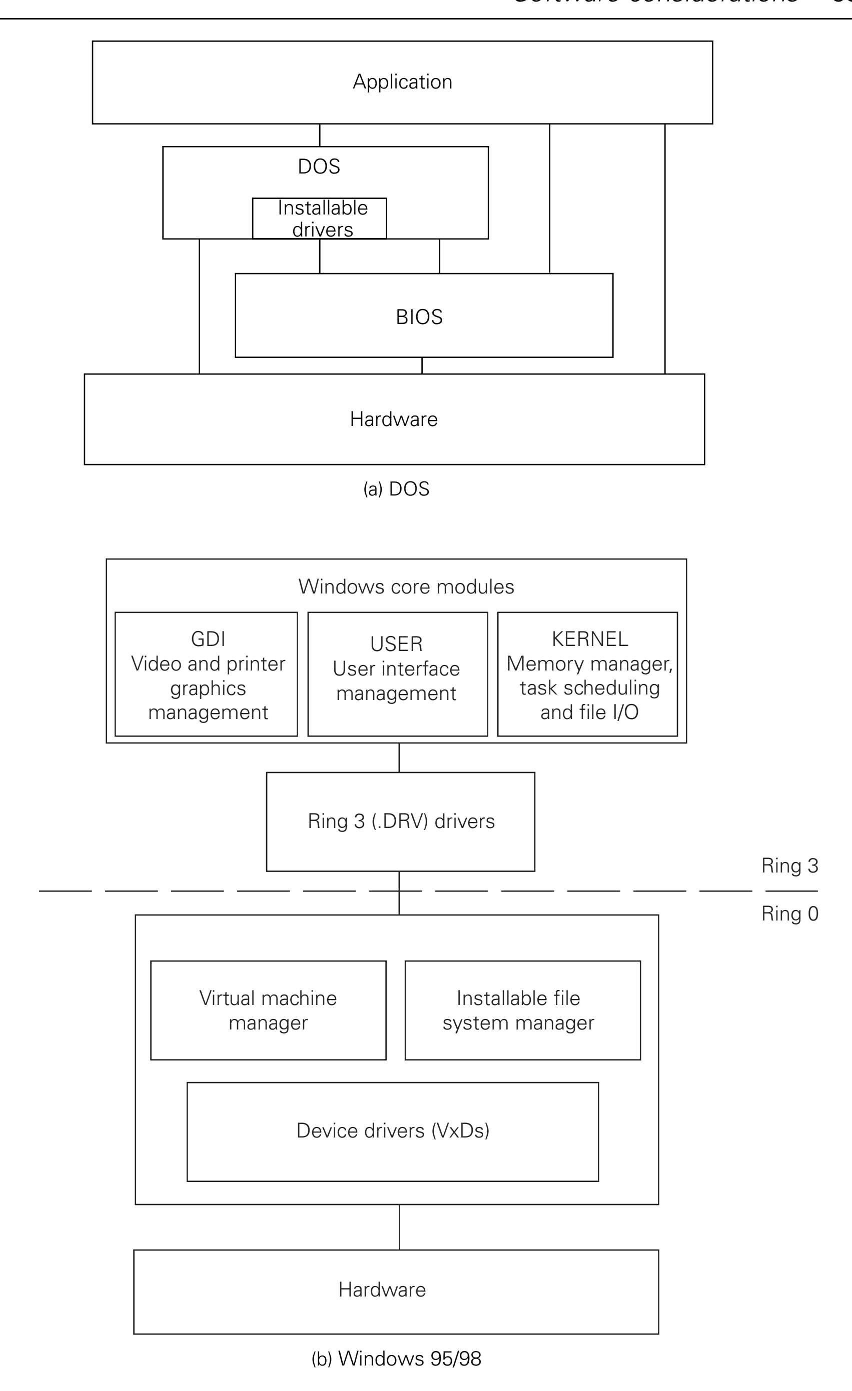
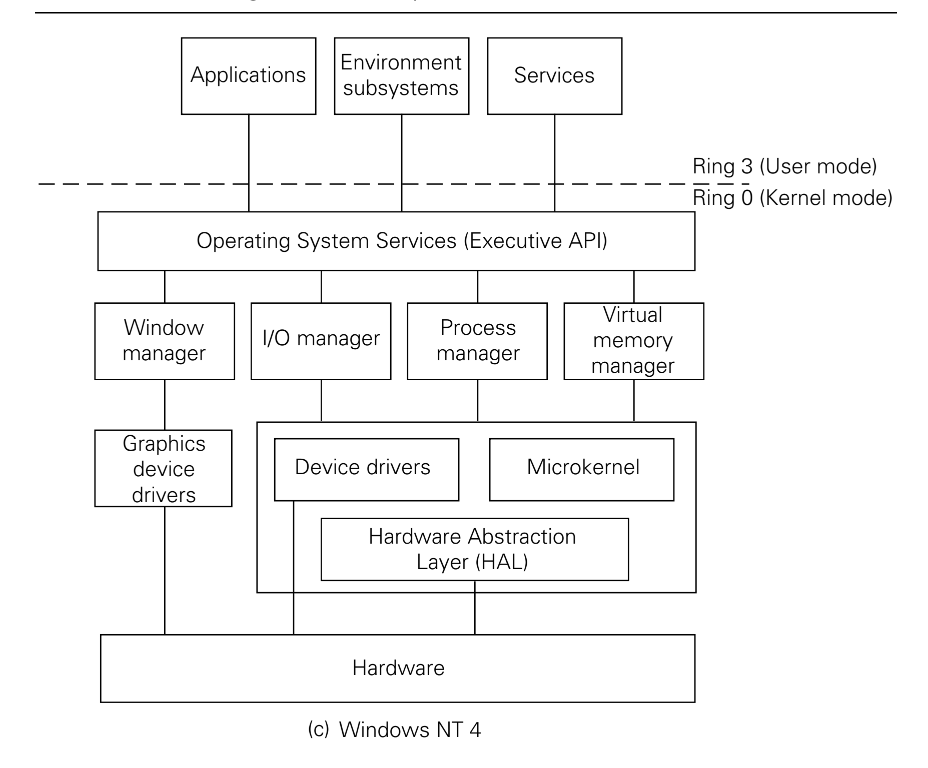


Figure 2.4 *Comparative architecture of DOS and Microsoft Windows*



(c) Windows NT 4

Figure 2.4 *(continued)*

Windows 98 enforce a greater degree of isolation between processes by effectively allocating them each a separate queue.

Virtual memory and demand paging

We have already introduced the concept of virtual memory which Windows uses to isolate applications from each other and from the operating system. Under this scheme, Windows allocates memory to each application in 4 KB blocks known as pages. Windows NT's Virtual Memory Manager and Windows 98's Virtual Machine Manager use the processor's page translation mechanism to manipulate the address of each page. In this way, it can, for example, appear to an application program that a set of pages occupies contiguous 4 KB blocks, when in fact they are widely separated in physical memory.

An application's address space is normally very much greater than the amount of physical memory in the system. A 32-bit address provides access to up to 4 GB of memory, but a moderately

well-specified PC might contain only 128 MB. If the memory requirements of the system exceed the total amount of physical memory installed, Windows will automatically swap memory pages out to disk. Those pages that have been in memory the longest will be saved to a temporary page file, freeing physical memory when required. If a program attempts to access a page that resides on the disk, the processor generates a page fault exception. Windows traps this and reloads the required page.

This process is known as *demand paging*. It is performed without the knowledge of the Ring 3 program and in a well-designed desktop application has no significant effect on performance, other than perhaps a slight reduction in speed. It does, however, have important consequences in real-time systems. It is generally very difficult (or impossible) to predict when a page fault will occur – particularly when the page fault might be generated by another process running on the system. Furthermore, swapping of pages to and from the disk can take an indeterminate length of time, increasing latencies to typically 10–20 ms (although this figure is not guaranteed). This is clearly unacceptable if a fast and deterministic real-time response is required.

Device drivers

In order to facilitate device-independent interfacing, Windows NT and Windows 95/98 employ a system of device drivers. The system used by Windows NT is complex and supports several types of device driver. Of most interest are the *kernel mode* drivers, which can directly access the PC's hardware and interrupt subsystem. Windows 95 and 98 use a less robust system of device drivers, which are known as *VxDs* (or Virtual Extended Drivers). Both types of driver operate in Ring 0. Within the driver it is possible to handle interrupts and perform high speed I/O predictably and independently of the host (Ring 3) program.

Even though VxDs and kernel mode drivers provide useful facilities for the DA&C programmer, they do not solve all of the problems of real-time programming under Windows. Real-time control is particularly difficult. In this type of system, acquired data must be processed by the host program in order that a control signal can be generated. As the host program runs in Ring 3, it is not possible for it to generate the required control signal within a guaranteed time. The mechanisms used for routing data between the driver and the host program can also introduce non-deterministic behaviour into the system.

Interrupt handling and latency

Interrupt latency is one of the most problematic areas under Windows. Latency times can be many times greater than in a comparable DOS-based application. They can also be much more difficult to predict. There are several reasons for this, although they are all associated to some degree with the virtualization and prioritization of the interrupt system, and with the multitasking nature of Windows.

To illustrate some of the problems we will consider interrupt handling under Windows NT. Interrupts are prioritized within a scheme of Interrupt Request Levels (IRQLs). This mirrors the 8259A PIC's IRQ levels, but the IRQL scheme serves additional functions within the operating system. When an interrupt occurs:

- Windows NT's Trap Handler saves the current machine context and then passes control to its Interrupt Dispatcher routine.
- The Interrupt Dispatcher raises the processor's IRQL to that of the interrupting device, which prevents it from responding to lower level interrupts. Processor interrupts are then re-enabled so that higher priority interrupts can be recognized.
- The Interrupt Dispatcher passes control to the appropriate Interrupt Service Routine (ISR), which will reside in a device driver or within Windows NT's kernel.
- The ISR will generally do only a minimum of processing, such as capturing the status of the interrupting device. By exiting quickly, the ISR avoids delaying lower priority interrupts for longer than necessary. Before terminating, the ISR may issue a request for a Deferred Procedure Call (DPC).
- Windows will subsequently invoke the driver's DPC routine (using the *software* interrupt mechanism). The DPC routine will then carry out the bulk of the interrupt processing, such as buffering and transferring data.

From the DA&C programmer's perspective, the difficulty with this is that the delay before invocation of the DPC routine is indeterminate. Furthermore, although interrupts are prioritized within the kernel, the queuing of DPC requests means that any priority information is lost. Interrupt-generated DPCs are invoked in the order in which the DPC requests were received. Thus handling a mouse interrupt, for example, can take precedence over an interrupt from a DA&C card or communications port. This arrangement makes for a more responsive user interface, but can have important consequences for a time-critical DA&C application.

Handling interrupts under Windows is a fairly complex and time-consuming process which, together with the potential for lengthy page-fault exceptions, greatly increases interrupt latency and has an

undesirable effect on determinism. It can be very difficult to predict the length of time before an interrupt request is serviced under Windows, because of the complex rerouting and handling processes involved.

Re-entrancy

Much of the code in the Windows 3.1 system is non-re-entrant and should not, therefore, be called directly from within an interrupt handler. Other techniques have to be used in cases where acquired data is to be processed by non-re-entrant operating system services. An interrupt handler contained within a VxD might, for example, read pending data from an I/O port, store it in a buffer and then issue a call-back request to Windows. At some later time, when it is safe to enter Windows' services, Windows will call the VxD back. When the VxD regains control, it knows that Windows must be in a stable state and so the VxD is free to invoke file I/O and other services in order to process the data which its interrupt handler had previously stored. Note that similar techniques may be used in simple DOS applications, although the call-back mechanism is not supported by MS-DOS and must be built into the application program itself.

The re-entrancy situation is somewhat better in the 32-bit environments of Windows NT and Windows 98, largely because re-entrant code is a prerequisite for pre-emptive multitasking. Note, however, that Windows 95/98 also contains a significant quantity of 16-bit code. Much of this originates from Windows 3.1 and is not re-entrant.

Windows and real-time operating systems

Most recent versions of Windows can be run in conjunction with specially designed real-time operating systems (RTOSs). The intention is to take advantage of the user interface capabilities of Windows while retaining the deterministic performance of a dedicated real-time operating system. This type of arrangement is useful for allowing Windows to handle application setup and display processes while the time-critical monitoring and control routines are run under the supervision of the real-time operating system. The interaction between Windows and an RTOS can be complex and only a very brief overview will be provided here.

RTOSs work in conjunction with Windows by taking advantage of the privilege levels provided by all post-80286 processors. Windows' kernel operates in Ring 0 (the highest privilege level). This gives it control of other processes and allows it to access all I/O and memory addresses.

The real-time operating system must also work at the highest privilege level. It does this by either relegating Windows to a lower level, while providing an environment for and responses to Windows to make it 'think' that it is operating in Ring 0, or by coexisting with Windows at the same privilege level. In the latter case the RTOS interfaces to Windows (in part) via its driver interface – i.e. by linking to Windows NT via its kernel mode driver interface or by existing in the form of a VxD under Windows 95/98. Indeed, under Windows 3.1, time-critical portions of data-acquisition software were sometimes coded as a VxD, guaranteeing it precedence over other processes.

Those parts of an application running under the RTOS operate in Ring 0. Consequently, some RTOSs do not provide the same degree of intertask memory protection as normally afforded by Windows. This can compromise reliability, allowing the whole system to be crashed by a coding error in just one task.

Developers have adopted very different approaches to producing RTOSs. Several different techniques can be used, even under the same version of Windows, but whatever method or type of RTOS is chosen, the result is essentially that threads running under the RTOS benefit from much lower interrupt latencies and a far greater degree of determinism.

Other 'desktop' operating systems

In addition to the various versions of Microsoft Windows, two other multitasking operating systems are worthy of mention: UNIX and OS/2. Although these include certain features which facilitate their use in real-time systems, they were designed with more heavyweight multitasking in mind. They possess many features that are necessary to safely execute multiple independent desktop applications.

UNIX has perhaps the longest history of any operating system. It was originally developed in the early 1970s by AT&T and a number of different implementations have since been produced by other companies and institutions. It was used primarily on mainframes and minicomputers, but for some time, versions of UNIX, notably XENIX and Linux, have also been available for microcomputers such as the PC.

In the PC environment, DOS compatibility was (and still is) considered to be of some importance. In general, UNIX can coexist with DOS on the PC allowing both UNIX and DOS applications to be run on the same machine. A common file system is also employed so that files can be shared between the two operating systems. DOS can also be run as a single process under UNIX in much the same way as

it is under Windows NT or Windows 98. UNIX itself is fundamentally a character-based system although a number of extensions and third-party shell programs provide powerful user interfaces and graphics support.

Of most interest, of course, is the applicability of UNIX to real-time processing. As already mentioned UNIX provides a heavyweight multitasking environment, the benefits of which have been discussed earlier. The UNIX kernel possesses a full complement of the features one would expect in such an environment: task scheduling, flexible priorities as well as interprocess communication facilities such as signals, queues and semaphores. In addition, UNIX provides extensive support for multiple users. Its network and communication features make it ideally suited to linking many processing sites. Typical industrial applications include distributed data acquisition and large-scale process control. UNIX also incorporates a number of quite sophisticated security features, which are particularly useful (if not essential) in applications such as factory-wide automation and control.

Some of the concepts behind UNIX have also appeared in subsequent operating systems. IBM's OS/2, for example, possesses many features which are similar to those offered by UNIX. The latest implementation for the PC, OS/2 Warp, was launched in 1994. This is a powerful 32-bit multiprocessing operating system which is well suited to complex multitasking on the PC. It requires only a modestly specified PC, provides support for Microsoft Windows applications and will multitask DOS applications with great efficiency.

Like UNIX, OS/2 provides comprehensive support for pre-emptive multitasking including dynamic priorities, message passing and semaphores for mutual exclusion of critical sections. OS/2 virtualizes the input/output system, but it also allows the programmer of time-critical applications and drivers to obtain the I/O privileges necessary for real-time use.

While both OS/2 and UNIX are extremely powerful operating systems, it should be remembered that many real-time applications do not require the degree of intertask protection and memory management provided by these environments. These desktop operating systems might, in some cases, be too complex and slow for real-time use. Nevertheless, they tend to be quite inexpensive when compared to more specialized RTOSs and are worth considering if robust multitasking is the primary concern.

Other real-time operating systems

We have already discussed versions of DOS and the BIOS designed for real-time use and have also mentioned RTOSs that are capable of

running in conjunction with Microsoft Windows. There are several other real-time operating systems on the market, such as Intel's iRMX, Microware OS/9000, Integrated Systems pSOSystem and QNX from QNX Software Systems Ltd. Unfortunately, space does not allow a detailed or exhaustive list to be presented. Note that most of these operating systems require an 80386 or later processor for optimum performance. Some are also capable of running MS-DOS and Windows (or special implementations of these operating systems), although, for the reasons described previously, this may result in a less deterministic system.

Summary

There are several options available to designers of real-time systems. Simple and relatively undemanding applications can often be accommodated by using MS-DOS, although this does not provide multitasking capabilities or the degree of determinism required by more stringent real-time applications. Microsoft Windows provides an even less deterministic solution, and interrupt latencies imposed by this environment can often be excessive. Various real-time operating systems (RTOSs) are also available, some of which are ROMable and suited for use in embedded applications. These include real-time versions of DOS and the BIOS, which can provide low interrupt latencies and efficient multitasking.

For many programmers, however, the choice of operating system for low and medium speed DA&C applications – particularly those which do not incorporate time-critical control algorithms – will be between MS-DOS and Windows. While Windows provides a far superior user interface, this benefit may be offset by poor interrupt latencies. DOS applications are generally somewhat simpler to produce and maintain, and it is often easier to retain a higher degree of control over their performance than with Windows programs. You should not underestimate the importance of this. To produce a reliable and maintainable system, it is preferable to employ the simplest hardware and operating system environment consistent with achieving the desired real-time performance. Only you, as the system designer or programmer, can decide which operating system is most appropriate for your own application.

In the remainder of this book, we will refrain from discussing characteristics of particular operating systems where practicable. Note, however, that the software listings provided in the following chapters were written for a real-mode DOS environment. If you intend to use them under other processor modes or operating systems, you should ensure that you adapt them accordingly.

2.4 Robustness, reliability and safety

Unreliable DA&C systems are, unfortunately, all too common. Failure of a DA&C system may result in lost time and associated expense or, in the case of safety-critical systems, even in injury or death! The quality of hardware components used will of course influence the reliability of the system. Of most practical concern in this book, however, is the reliability of DA&C *software*. This is often the most unreliable element of a DA&C system especially during the time period immediately following installation or after subsequent software upgrades. Several development techniques and methodologies have been developed in order to maximize software reliability. These generally impose a structured approach to design, programming and testing, and include techniques for assessing the complexity of software algorithms. These topics are the preserve of software engineering texts and will not be covered here. It is impracticable to cover every factor that you will need to consider when designing DA&C software, and the following discussion is confined to a few of the more important general principles of software development, testing and reliability as they relate to DA&C. Interested readers should consult Maguire (1993), Bell *et al.* (1992) or other numerous software engineering texts currently on the market for further guidance.

Software production and testing

The reliability of a DA&C system is, to a great extent, determined by the quality of its software component. Badly written or inadequately tested software can result in considerable expense to both the supplier and the end user, particularly where the system plays a critical role in a high volume production process.

As we have already noted, an important requirement for producing correct, error-free and, therefore, reliable programs is simplicity. The ability to achieve this is obviously determined to a large extent by the nature of the application. However, a methodical approach to software design can help to break down the problem into simpler, more manageable, portions. The value of time spent on the design process should not be underestimated. It can be very difficult to compensate for design flaws discovered during the subsequent coding or testing stages of development.

Perhaps the most important step when designing a DA&C program (or indeed any type of program) is to identify those elements of the software that are critical for correct functioning of the system. These

often occupy a relatively small proportion of a DA&C program. They might, for example, include monitoring and control algorithms or routines for warning the operator of error conditions. Isolating critical routines in this way permits a greater degree of effort to be directed toward the most important elements of the program and thus allows optimal use to be made of the available development time.

Libraries

A common means of reducing the development effort needed for non-critical software, thus enabling resources to be concentrated on the most critical routines, is to make use of pre-written software libraries. The user interface, for example, often occupies a high proportion of the total software development time, and this may be reduced by using appropriate tools. A number of C and Pascal user-interface libraries are currently on the market. These allow a standardized user interface to be incorporated into the software. As the library routines are generally well tested and normally include thorough range checking, validation, and error trapping facilities, this also helps to reduce the incidence of coding errors.

Dedicated DA&C libraries, such as those included with National Instruments' LabWindows/CVI, provide support for real-time graphical displays and virtual instruments such as digital voltmeters and oscilloscopes. Drivers for RS-232, IEEE-488, and a range of DA&C cards might also be supplied, particularly in libraries provided by manufacturers of DA&C hardware. Tools for post-acquisition analysis of data may be included as well. Typically, these incorporate a range of facilities, from simple arithmetic array operations to support for complex signal processing (e.g. fast Fourier transforms, filtering and signal generation). Many libraries are oriented towards development of Windows programs, although some provide a degree of portability between environments.

One of the most important points to bear in mind when selecting a library is the availability of source code. Some libraries are supplied only in compiled object file format. This obviously limits the degree to which the system can be adapted to a client's needs.

Testing

Thorough testing is essential to ensure that each routine behaves as expected when subjected to every possible combination of inputs. In all but the simplest DA&C systems, this is usually facilitated by testing each program module independently of the others. In this way, the inputs supplied to each routine can be precisely controlled

in order to ensure that all possible code paths are executed. This procedure usually involves supplying extreme or over-range inputs, which the routine should never receive in a correctly functioning system. Critical routines in particular should be designed to trap erroneous inputs without propagating the error on to other code modules.

Modular testing can be difficult to achieve in time-dependent DA&C systems. This is particularly so in routines that measure elapsed time or which check for timeouts in dynamic systems. The behaviour of such a routine might vary depending upon the times at which certain inputs are applied. In order to ensure that the dynamic behaviour of the system can be adequately modelled during testing, it may be necessary to build a complete *test harness*. This consists of a hardware interface together with software support routines, which provide a controlled environment for the module under test. Test harnesses may range from a simple bank of lamps or switches designed to monitor the states of digital I/O lines, to a complex suite of test programs or even to a dedicated test computer. They may also incorporate items of test equipment such as logic analysers and digital storage oscilloscopes.

When performing time-dependence tests, allowances should be made for any variations in timing that might occur in a fully working system. These variations might arise from changes in the system's loading conditions or from occasional replacement of some system component by a faster variant. It is generally good practice to avoid making one routine dependent on the timing of some other routine or hardware subsystem. There is, of course, a limit to how far this requirement can be implemented in practical DA&C applications. Sufficient latitude should be built into the system (e.g. by buffering data) to accommodate both transient and persistent variations in timing.

When all modules have been independently tested, they should be gradually combined and further checks performed to ensure that there are no unforeseen interactions between them. Again, thorough timing tests may have to be carried out, possibly with the aid of a suitable test harness. Testing and optimization can also be facilitated by using profiling techniques which accurately measure the proportion of time spent executing each section of code.

Assertions

Coding errors can cause software to fail in one of two ways. The failure may be immediately obvious resulting in, for example, a corrupted display, a malfunctioning control system or the termination of a DA&C program. Alternatively, the consequences of a failure may be

more subtle, causing, for example, only a slight degradation in the performance of a control system. These two classes of software failure are sometimes, rather confusingly, termed hard and soft failures.

Hard failures are greatly preferable, simply because they are immediately obvious to the user. Although soft failures are more subtle, their consequences can ultimately be no less serious. Indeed they may be much worse. As the user will probably be unaware of any problem, soft failures can go undetected for long periods. Hard failures are generally the cheapest to rectify as most are detected during the development and testing phase, prior to delivery of the software.

What is needed is a way to convert insidious soft failures and latent software errors into hard failures. Assertions are invaluable for this purpose. These are simply software statements (actually macros in C and C++) which terminate execution of the program if their argument is FALSE or zero. Generally the argument of an assertion is a logical expression that defines a set of acceptable conditions at some point within the program. These conditions often denote permissible ranges of selected variables. The argument of the assertion must evaluate to TRUE (or 1) if all conditions are met, in which case the program proceeds as normal. When an assertion fails, however, the program is halted and the location of the failed assertion is displayed on screen.

Assertions can be used at virtually any point within the code. Remember though that they are suitable only to detect coding errors and situations that *should* never occur within your program. They should not be used to trap legitimate error conditions such as a serial communications error or printer out-of-paper error. Assertions tend to be used most frequently to range check function arguments and function return values. An example of an assertion statement in C is shown in the following code fragment.

```
double VMax;                    /* Maximum input */
double VMin;                    /* Minimum input */

void CalcPID(double V, double T, double *Y)
{
ASSERT ((V < VMax) && (V > VMin) && (T >= 0)); /* Range check V and T */

/* Function body: calculates result, Y, based on arguments V and T */

}
```

Most C and C++ compilers include an ASSERT macro. Code generation within the ASSERT macro is controlled by the Debug compiler option (or equivalent compiler define) allowing executable assertion code to be generated only during development. Prior to delivery of

the software, assertions can be compiled out so that no performance overheads are incurred in the final build.

System monitoring and error checks

The reliability of a working DA&C system can often be improved by incorporating facilities for automatic self-testing. Such facilities might be used to periodically test the status of hardware components or to check the integrity of software modules. The PC's BIOS executes a number of self-test routines when the computer is started up. These Power On Self Test (POST) routines include checks to ensure that none of the memory locations are faulty and to verify that the keyboard and disk subsystems are working correctly. It may be advisable to incorporate similar test routines within your DA&C applications in order to check that data-acquisition cards or data-logging units are operating normally. These test routines might run automatically when the system is first started and, perhaps, periodically thereafter.

Tests that can usually be performed on start-up include those that check for the presence of adaptor cards or that confirm the integrity of communications links. It may also be necessary to ensure that all subsystems on which the DA&C program is reliant (e.g. PLCs or intelligent data loggers) are operational and on line. In long-term data-logging applications, where the system might have to run unattended, it is prudent to verify that all other essential peripherals (e.g. printer) are connected and correctly configured before data logging commences.

In applications that require a high degree of operator intervention it might be desirable to give the user some control over when and how the tests are performed. Such an approach provides greater flexibility but does require a higher level of operator skill. Certain checks, such as monitoring and correcting for zero drift in signal-conditioning circuits (see Chapter 9) may, in many cases, have to be carried out manually. Others tend to be more amenable to automation. Even if certain checks cannot be automated, it may still be possible to incorporate routines which will prompt the operator when activities such as rezeroing or recalibration are overdue.

Range checking inputs and outputs

One of the most important safety features that can be built into any program is a comprehensive system of range checking. A DA&C program must be able to handle unexpectedly large or small data arriving at its inputs. This necessitates writing extensive checking and validation routines to handle user-supplied data as well as

data acquired from sensors. By maintaining all inputs within an acceptable range, it is possible to guard against problems such as numeric overflows which, if undetected, can cause the system to fail unpredictably.

Out-of-range data may arise as a result of factors such as electrical noise, a faulty or inadequately calibrated sensor, or the failure of some external subsystem. It might be possible to ignore or suppress transient faults such as those due to electrical noise, although if they occur frequently, they could be indicative of a more persistent problem or of an inherent design fault. Techniques, such as filtering and hysteresis, which can make the system more immune to the effects of noise and transient fluctuations, are described in Chapter 4.

It is usually preferable to integrate range-checking code into the routines that are responsible for inputting data into the system. This reduces the likelihood that any erroneous data will be passed on to other elements of the software. Range checking may also be necessary at a number of other critical points within the program. The acceptable range of values that each item of data is allowed to take might be fixed throughout the execution of the program, or it might vary dynamically depending upon other inputs or upon the values of previous readings. When thoroughly implemented, range-checking and validation routines will normally make up a considerable proportion of the whole program. Bear in mind though that the requirement for range checking, if enforced too rigorously, can impose an unacceptable performance penalty and should always be applied with discretion.

Status checks

When the PC has to communicate with one or more external units (e.g. remote data loggers, PLCs or other computers), it can be useful for each unit to provide some form of status indication. This allows the PC to determine whether each external device is functioning correctly. Typically status indicators consist of simple digital signals controlled via relays or switches. These should usually be configured to operate in the so-called *fail-safe* mode (see Chapter 3).

Other status-verification techniques can be used in some cases. The PC might repeatedly poll each external unit to determine whether it is on line. Properly functioning units would acknowledge the poll by generating a suitable signal. The polling procedure might be incorporated into routines which initialize the unit or which regularly interrogate it. This type of approach can be used on multi-drop bus-based systems: for example, an RS-485 network of signal-conditioning modules. A similar, alternative method requires one element of the DA&C system to issue a periodic *heartbeat* signal.

This is continuously monitored by other system components, which might then be required to respond within predefined time limits. Any interruptions in the periodic signal would indicate the failure of some component or a faulty communications link. Periodic signals can also be used to refresh dedicated monitoring circuitry, such as watchdog timers. These systems notify the PC if the periodic refresh signal from an external unit fails to arrive on time.

Responding to faults

When a fault is detected, its severity and nature (e.g. whether the fault is transient, intermittent or persistent) should be assessed. A decision must also be made as to whether the system can continue to function reliably, albeit with a reduced functionality. This decision may be made in advance by the system designer and hard-coded into the DA&C software. Alternatively, it might be left to the operator to decide what actions should be taken in the context of specific faults.

In either case it is important for the system to display appropriate error or warning messages. Messages should be clear and precise. Although numeric error codes can help to identify a particular error, they should always be accompanied by an informative description of the error. It is often useful to include a suggestion of any remedial action that might have be taken by the operator. On-screen error messages will be of little or no value in systems intended for long periods of unattended operation. In these cases, it can be useful for the PC to record operational faults on some form of permanent storage device such as a hard disk or printer. The nature of the fault, the date and time that it occurred, and any relevant conditions prevailing at the time should also be logged in order to aid subsequent fault tracing and diagnosis.

A fault or error may be detected at any one of many possible points within the hierarchical function structure of a program. Faults are often detected in interface and driver routines, which typically reside at the lower levels in the structure. Error codes or flags then usually have to be passed back up the structure to be handled (e.g. recorded) by higher level routines. Although this tends to allow the programmer to create a well-structured and tidy code, it requires a degree of care. Once an error or fault has occurred it is possible that it might then also trigger a stream of errors in related routines, which must be handled in a well-defined and consistent manner.

It is essential to adopt a systematic and adaptable method of error handling. One solution is to assign each possible error condition a unique 8-bit or 16-bit integer code. The code should be unique to the routine which detected the error and should also indicate the *type* of error that it represents. As soon as an error is detected, an

error-recording routine should be called. This might store the error code in a queue or buffer and set a flag to indicate that one or more errors have occurred. Control should then be returned through the function hierarchy to a high level error handling routine, which can then process any pending errors. In this type of error-handling model, there will be a delay between recognition of the fault and a subsequent response. The system designer must assess this delay and decide whether it is acceptable within the time constraints imposed by the software specification.

The course of action taken in response to a fault will be highly dependent upon the nature of the application. Many faults will be minor ones that can be rectified by requesting the operator to make some adjustment to the system. Other faults can be more serious, leaving the system in an unstable or inoperable state. The software should, in these cases, shut the system down in a safe and orderly manner. Certain faults can be catastrophic, causing complete failure of the DA&C program and/or the PC on which it is running. Although the programmer should take whatever precautions are necessary to ensure that the system will provide a controlled response, there is little that can be done to prevent hardware problems such as a disk failure, loss of power or electrostatic discharge.

PCs and the software running on them are very complex systems and there are numerous ways in which they can fail. The potential for failure of both hardware and software should be considered. Many failure modes can be catastrophic and will result in complete failure of monitoring and control systems. Because of this, PC-based systems and software should not be relied upon to oversee safety-critical processes without using appropriate backup mechanisms to ensure total safety. Indeed, the information presented in this book is not intended for use in safety-critical applications. If you use it in such, you do so at your own risk. You are advised to cross-check each item of information which you use in your software with independent sources. You should also thoroughly test all program code that you use, regardless of its source, to ensure that it works correctly and reliably under the specific conditions of your application.

Part 2 Sampling Fundamentals

This Page Intentionally Left Blank

3 Sensors and interfacing

Hardware characteristics such as non-linearity, response times and susceptibility to noise can have important consequences in a data-acquisition system. They often limit performance and may necessitate countermeasures to be implemented in software. A detailed knowledge of the transfer characteristics and temporal performance of each element of the DA&C system is a prerequisite for writing reliable interface software. The purpose of this chapter is to draw your attention to those attributes of sensors, actuators, signal conditioning and digitization circuitry that have a direct bearing on software design. While precise details are generally to be found in manufacturer's literature, the material presented in the following sections highlights some of the fundamental considerations involved. Readers are referred to Eggebrecht (1990) or Tompkins and Webster (1988) for additional information.

3.1 Introduction

DA&C involves measuring the parameters of some physical process, manipulating the measurements within a computer, and then issuing signals to control that process. Physical variables such as temperature, force or position are measured with some form of sensor. This converts the quantity of interest into an electrical signal which can then be processed and passed to the PC. Control signals issued by the PC are usually used to drive external equipment via an actuator such as a solenoid or electric motor.

Many sensors are actually types of transducer. The two terms have different meanings, although they are used somewhat interchangeably in some texts. Transducers are devices that convert one form of energy into another. They encompass both actuators and a subset of the various types of sensor.

Signal types

The signals transferred in and out of the PC may each be one of two basic types: analogue or digital. All signals will generally vary in time. In changing from one value to another, analogue signals vary smoothly (i.e. continuously), always assuming an infinite sequence of intermediate values during the transition. Digital signals, on the other hand, are discontinuous, changing only in discrete steps as shown in Figure 3.1.

Digital data are generally stored and manipulated within the PC as binary integers. As most readers will know, each binary digit (bit) may assume only one of two states: low or high. Each bit can, therefore, represent only a 0 or a 1. Larger numbers, which are needed to represent analogue quantities, are generally coded as

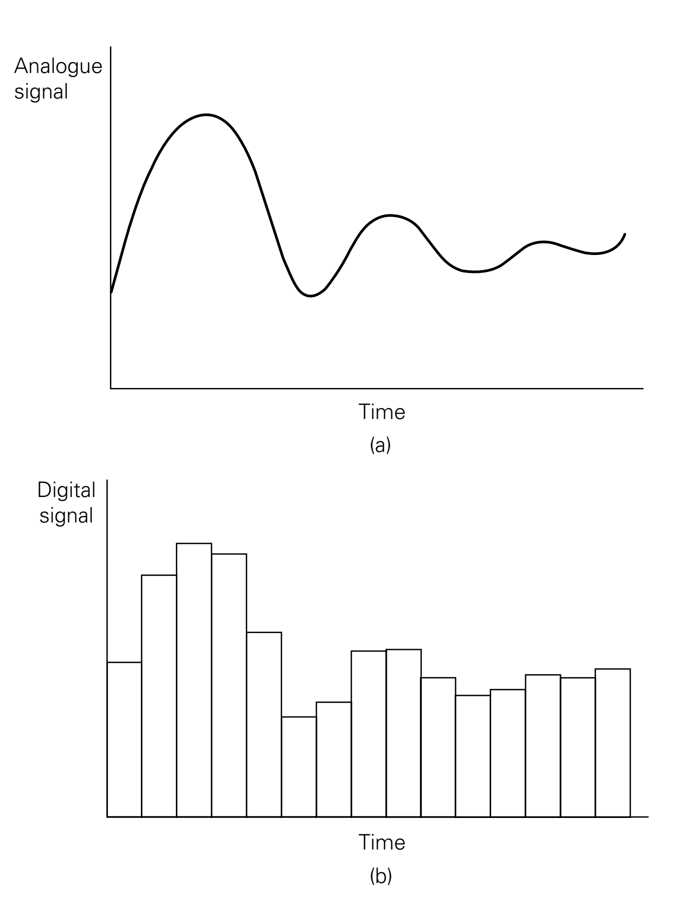


Figure 3.1 *Diagram contrasting (a) analogue and (b) digital signals*

combinations of typically 8, 12 or 16 bits. Binary numbers can only change in discrete steps equal in size to the value represented by the least significant bit (LSB). Because of this, binary (i.e. digital) representations of analogue signals cannot reflect signal variations smaller than the value of the LSB. The principal advantage of digital signals is that they tend to be less susceptible than their analogue counterparts to distortion and noise. Given the right communication medium, digital signals are more suited to long-distance transmission and to use in noisy environments.

Pulsed signals are an important class of digital signals. From a physical point of view, they are basically the same as single-bit digital signals. The only difference is in the way in which they are applied and interpreted. It is the static bit patterns (the presence, or otherwise of certain bits) that are the important element in the case of digital signals. Pulsed signals, on the other hand, carry information only in their timing. The frequency, duration, duty cycle or absolute number of pulses are generally the only significant characteristics of pulsed signals. Their amplitude does not carry any information.

Analogue signals carry information in their magnitude (level) or shape (variation over time). The shape of analogue signals can be interpreted either in the time or frequency domain. Most 'real-world' processes that we might wish to measure or control are intrinsically analogue in nature.

It is important to remember, however, that the PC can read and write only digital signals. Some sensing devices, such as switches or shaft encoders, generate digital signals which can be directly interfaced to one of the PC's I/O ports. Certain types of actuator, such as stepper motors or solenoids, can also be controlled via digital signals output directly from the PC. Nevertheless, most sensors and actuators are purely analogue devices and the DA&C system must, consequently, incorporate components to convert between analogue and digital representations of data. These conversions are carried out by means of devices known as analogue-to-digital converters (ADCs) or digital-to-analogue converters (DACs).

Elements of a DA&C system

A typical PC-based DA&C system might be designed to accept analogue inputs from sensors as well as digital inputs from switches or counters. It might also be capable of generating analogue and digital outputs for controlling actuators, lamps or relays. Figure 3.2 illustrates the principal elements of such a system. Note that, for clarity, this figure does not include control signals. You should bear in mind that, in reality, a variety of digital control lines will be

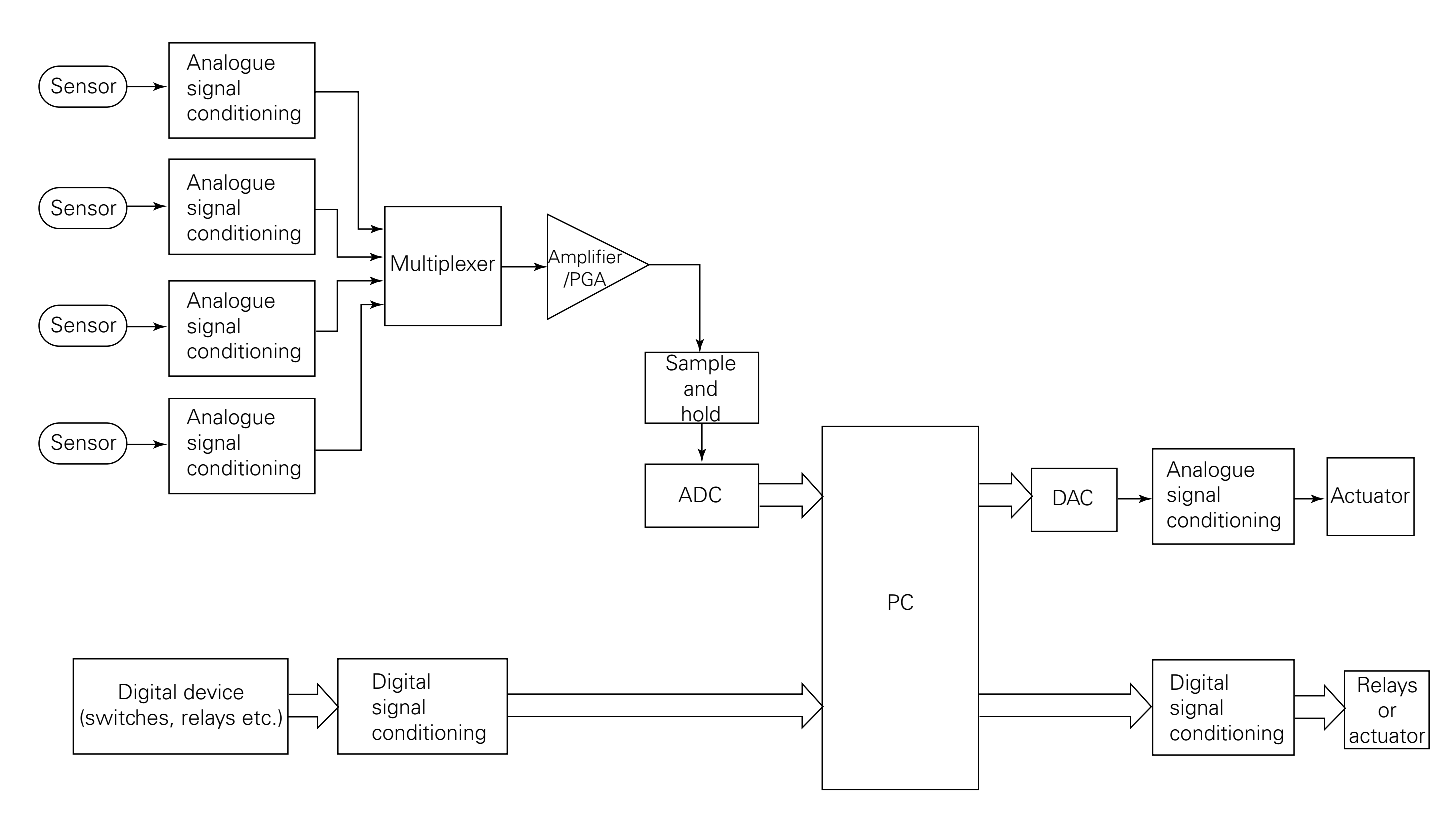


Figure 3.2 *A typical PC-based DA&C system*

required by devices such as multiplexers, programmable-gain amplifiers and ADCs. Depending upon the type of system in use, the device generating the control signals may be either the PC itself or dedicated electronic control circuitry.

The figure shows four separate component chains representing analogue input, analogue output, digital input and digital output. An ADC and DAC shown in the analogue I/O chains facilitate conversion between analogue and digital data formats.

Digital inputs can be generated by switches, relays or digital electronic components such as timer/counter ICs. These signals usually have to undergo some form of digital signal conditioning, which might include voltage level conversion, isolation or buffering, before being input via one of the PC's I/O ports. Equally, low level digital outputs generated by the PC normally have to be amplified and conditioned in order for them to drive actuators or relays.

A similar consideration applies to analogue outputs. Most actuators have relatively high current requirements which cannot be satisfied directly by the DAC. Amplification and buffering (implemented by the signal conditioning block) is, therefore, usually necessary in order to drive motors and other types of actuator.

The analogue input chain is the most complex. It usually incorporates not only signal-conditioning circuits, but also components such as a multiplexer, programmable-gain amplifier (PGA) and sample-and-hold (S/H) circuit. These devices are discussed later in this chapter. The example shown is a four-channel system. Signals from four sensors are conditioned and one of the signals is selected by the multiplexer under software control. The selected signal is then amplified, and digitized before being passed to the PC.

The distinction between elements in the chain is not always obvious. In many real systems the various component blocks are grouped within different physical devices or enclosures. To minimize noise, it is common for the signal-conditioning and preamplification electronics to be separated from the ADC and from any other digital components. Although each analogue input channel has only one signal-conditioning block in Figure 3.2, this block may, in reality, be physically distributed along the analogue input chain. It might be located within the sensor or at the input to the ADC. In some systems, additional components are included within the chain, or some elements, such as the S/H circuit, might be omitted.

The digital links in and out of the PC can take a variety of forms. They may be direct (although suitably buffered) connections to the PC's expansion bus, or they may involve serial or parallel transmission of data over many metres. In the former case, the ADC, DAC and associated interface circuitry are often located on I/O

cards which can be inserted in one of the PC's expansion bus slots or into a PCMCIA slot. In the case of devices which interface via the PC's serial or parallel ports, the link is implemented by appropriate transmitters, bus drivers and interface hardware (which are not shown in Figure 3.2). Data transfer techniques and the various types of I/O interface devices available are discussed in Chapters 6 to 8.

3.2 Digital I/O

Digital (including pulsed) signals are used for interfacing to a variety of computer peripherals as well as for sensing and controlling DA&C devices. Some sensing devices such as magnetic reed switches, inductive proximity switches, mechanical limit switches, relays or digital sensors, are capable of generating digital signals which can be read into the PC. The PC may also *issue* digital signals for controlling solenoids, audio-visual indicators or stepper motors. Digital I/O signals are also used for interfacing to digital electronic devices such as timer/counter ICs or for communicating with other computers and Programmable Logic Controllers (PLCs).

Digital signals may be encoded representations of numeric data or they may simply carry control or timing information. The latter are often used to synchronize the operation of the PC with external equipment using periodic clock pulses or handshaking signals. Handshaking signals are used to inform one device that another is ready to receive or transmit data. They generally consist of level-active, rather than pulsed, digital signals and, as we shall see in Chapters 7 and 8, they are essential features of most parallel and serial communication systems. Pulsed signals are not only suitable for timing and synchronization: they are also often used for event counting or frequency measurement. Pulsed inputs, for pacing or measuring elapsed time, can be generated either by programmable counter/timer ICs on plug-in DA&C cards or by programming the PC's own built-in timers. Pulsed inputs are often used to generate interrupts within the PC in response to specific external events.

TTL-level digital signals

Transistor–transistor logic (TTL) is a type of digital signal characterized by nominal 'high' and 'low' voltages of +5 V and 0 V. TTL devices are capable of operating at high speeds. They can switch their outputs in response to changing inputs within typically 20 ns and can deal with pulsed signals at frequencies up to several tens of MHz. TTL devices can also be directly interfaced to the PC. The main problem

with using TTL signals for communicating with external equipment is that TTL ICs have a limited current capacity and are suitable for directly driving only low current (i.e. a few milliamps) devices such as other TTL ICs, LEDs and transistors. Another limitation is that TTL is capable of transmission over only relatively short distances. While it is ideal for communicating with devices on plug-in DA&C cards, it cannot be used for long-distance transmission without using appropriate bus transceivers.

The PC's expansion bus, and interface devices such as the Intel 8255 Programmable Peripheral Interface (PPI), provide TTL-level I/O ports through which it is possible to communicate with peripheral equipment. Many devices that generate or receive digital level or pulsed signals are TTL compatible and so no signal conditioning circuits, other than perhaps simple bus drivers or tristate buffers, are required. Buffering, optical isolation, electromechanical isolation and other forms of digital signal conditioning may be needed in order to interface to remote or high current devices such as electric motors or solenoids.

Digital signal conditioning and isolation

Digital signals often span a range of voltages other than the 0 to 5 V encompassed by TTL. Many pulsed signals are TTL compatible, but this is not always true of digital *level* signals. Logic levels higher or lower than the standard TTL voltages can easily be accommodated by using suitable voltage attenuating or amplification components. Depending upon the application, the way in which digital I/O signals are conditioned will vary. Many applications demand a degree of isolation and/or current driving capability. The signal-conditioning circuits needed to achieve this may reside either on digital I/O interface cards which are plugged into the PC's expansion bus or they may be incorporated within some form of external interface module. Interface cards and DA&C modules are available with various degrees of isolation and buffering. Many low cost units provide only TTL-level I/O lines. A greater degree of isolation and noise immunity is provided by devices which incorporate optical isolation and/or mechanical relays.

TTL devices can operate at high speeds with minimal propagation delay. Any time delays that may be introduced by TTL devices are generally negligible when compared with the execution time of software I/O instructions. TTL devices and circuits can thus be considered to respond almost instantaneously to software `IN` and `OUT` instructions. However, this is not generally true when additional isolating or conditioning devices are used. Considerable delays can

result from using relays in particular, and these must be considered by the designer of the DA&C software.

Opto-isolated I/O

It is usually desirable to electrically isolate the PC from external switches or sensors in order to provide a degree of overvoltage and noise protection. Opto-isolators can provide isolation from typically 500 V to a few kV at frequencies up to several hundred kHz. These devices generally consist of an infrared LED optically coupled to a phototransistor within a standard DIL package as shown in Figure 3.3. The input and output parts of the circuit are electrically isolated. The digital signal is transferred from the input (LED) circuit to the output (phototransistor) by means of an infrared light beam. As the input voltage increases (i.e. when a logical high level is applied), the photodiode emits light which causes the phototransistor to conduct. Thus the output is directly influenced by the input state while remaining electrically isolated from it.

Some opto-isolating devices clean and shape the output pulse by means of built-in Schmitt triggers. Others include Darlington transistors for driving medium current loads such as lamps or relays. Mains and other AC loads may be driven by solid state relays which are basically opto-isolators with a high AC current switching capability.

Opto-isolators tend to be quite fast in operation, although somewhat slower than TTL devices. Typical switching times range from

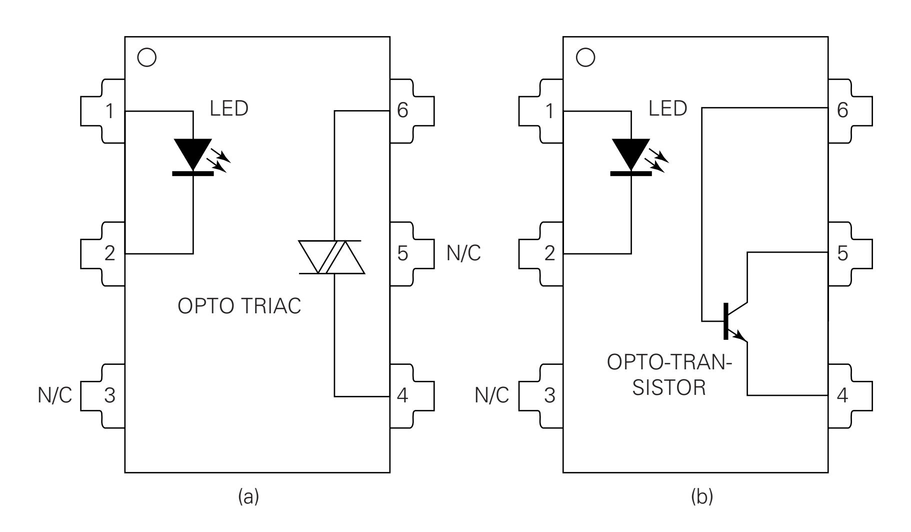


Figure 3.3 *Typical opto-isolator DIL packages: (a) an opto-triac suitable for mains switching, and (b) a simple opto-transistor device*

about 3 μs to 100 μs, allowing throughputs of about 10–300 Kbit/s. Because of their inherent isolation and slower response times, opto-isolators tend to provide a high degree of noise immunity and are ideally suited to use in noisy industrial environments. To further enhance rejection of spurious noise spikes, opto-isolators are sometimes used in conjunction with additional filtering and pulse-shaping circuits. Typical filters can increase response times to, perhaps, several milliseconds. It should be noted that opto-couplers are also available for isolating analogue systems. The temporal response of any such devices used in analogue I/O channels should be considered as it may have an important bearing on the sampling rate and accuracy of the measuring system.

Mechanical relays and switches

Relays are electromechanical devices which permit electrical contacts to be opened or closed by small driving currents. The contacts are generally rated for much larger currents than that required to initiate switching. Relays are ideal for isolating high current devices, such as electric motors, from the PC and from sensitive electronic control circuits. They are commonly used on both input and output lines. A number of manufacturers provide plug-in PC interface cards with typically 8 or 16 PCB-mounted relays. Other digital output cards are designed to connect to external arrays or racks of relays.

Most relays on DA&C interface cards are allocated in arrays of 8 or 16, each one corresponding to a single bit in one of the PC's I/O ports. In many (but not all) cases a high bit will energize the relay. Relays provide either normally open (NO) or normally closed (NC) contacts or both. NO contacts remain open until the relay coil is energized, at which point they close. NC contacts operate in the opposite sense. Ensure that you are aware of the relationship between the I/O bit states and the state of the relay contacts you are using. It is prudent to operate relays in fail-safe mode, such that their contacts return to an inactive (and safe) state when de-energized. Exactly what state is considered inactive will depend upon the application.

Because of the mass of the contacts and other mechanical components, relay switching operations are relatively slow. Small relays with low current ratings tend to operate faster than larger devices. Reed relays rated at around 1 A, 24 V (DC) usually switch within about 0.25 to 1 ms. The operating and release times of miniature relays rated at 1 to 3 A usually fall in the range from about 2 to 5 ms. Larger relays for driving high power DC or AC mains loads might take up to 10 or 20 ms to switch. These figures are intended only as

rough guidelines. You should consult your hardware manufacturer's literature for precise switching specifications.

Switch and relay debouncing

When mechanical relay or switch contacts close, they tend to vibrate or bounce for a short period. This results in a sequence of rapid closures and openings before the contacts settle into a stable state. The time taken for the contacts to settle (known as the bounce time) may range from a few hundred microseconds for small reed relays up to several milliseconds for high power relays. Because bouncing relay contacts make and break several times, it can appear to the software monitoring the relay that several separate switching events occur each time the relay is energized or de-energized. This can be problematic, particularly if the system is designed to generate interrupts as a result of each contact closure.

There are two ways in which this problem can be overcome: hardware debouncing and software debouncing. The hardware method involves averaging the state of the switch circuit over an interval of a few milliseconds so that any short-lived transitions are smoothed out and only a gradual change is recorded. A typical method is to use a resistor/capacitor (RC) network in conjunction with an inverting Schmitt buffer. Tooley (1995) discusses hardware debouncing in more detail and illustrates several simple debouncing circuits.

The software debouncing technique is suitable only for digital *inputs* driven from relays and switches. It cannot of course be applied to relay signals *generated* by the PC. The technique works by repeatedly reading the state of the relay contact. The input should be sensed at least twice and a time delay sufficient to allow the contacts to settle should be inserted between the two read operations. If the state of the contacts is the same during both reads, that state is recorded. If it has changed, further delays and read operations should be performed until two successive read operations return the same result. An appropriate limit must, of course, be imposed on the number of repeats that are allowed during the debounce routine in order to avoid the possibility of unbounded software loops. Listing 3.1 illustrates the debouncing technique. It assumes that the state of the relay contacts is indicated by bit 0 of I/O port 300h. The routine exits with a non-zero value in CX and the debounced relay state in bit 0 of AL. If the relay does not reach a steady state after four read operations (i.e. three delay periods), CX contains zero to indicate the error condition. The routine can easily be adapted to deal with a different bit or I/O port address.

The delay time between successive read operations (implemented by the `DBDelay` subroutine which is not shown) should be chosen to be

Listing 3.1 *Contact debouncing algorithm*

```
         mov  dx,300h  ;Port number 300h for sensing relay
         mov  cx,4     ;Initialize timeout counter

DBRead:  in   al,dx    ;Read relay I/O port
         and  al,01h   ;Isolate relay status bit (bit 0)

         cmp  cx,4     ;Is this the first read ?
         je   DBLoop   ; - Yes, do another
         cmp  al,bl    ; - No, was relay the same as last time ?
         je   DBExit   ;    - Yes, relay in steady state so exit

DBLoop:  mov  bl,al    ;Store current relay state
         call DBDelay  ;Do delay to allow relay contacts to settle
         loop DBRead   ;Read again, unless timed out

DBExit:
```

just long enough to encompass the maximum contact bounce period expected. For most mechanical switches, this will be typically several milliseconds (or even tens of milliseconds for some larger devices). As a rough rule-of-thumb, the smaller the switch (i.e. the lower the mass of the moving contact) the shorter will be the contact bounce period. In choosing the delay time, remember to take account of the time constant of any other circuitry that forms part of the digital input channel.

Listing 3.1 is not totally foolproof: it will fail if the contact bounce period exactly coincides with the time period between samples. To improve the efficiency of this technique, you may wish to adapt Listing 3.1 in order to check that the final relay state actually remains stable for a number of consecutive samples over an appropriate time interval.

3.3 Sensors for analogue signals

Sensors are the primary input element involved in reading physical quantities (such as temperature, force or position) into a DA&C system. They are generally used to measure analogue signals although the term ‘sensor’ does in fact encompass some digital devices such as proximity switches. In this section we will deal only with sensing analogue signals.

Analogue signals can be measured with sensors that generate either analogue or digital representations of the quantity to be measured (the measurand). The latter are often the simplest to interface to the PC as their output can be read directly into one the PC’s

I/O ports via a suitable digital input card. Examples of sensors with digital outputs include shaft encoders and some types of flow sensor.

Most types of sensor operate in a purely analogue manner, converting the measurand to an equivalent analogue signal. The sensor output generally takes the form of a change in some electrical parameter such as voltage, current, capacitance or resistance. The primary purpose of the analogue signal-conditioning blocks shown in Figure 3.2 is to precondition the sensors' electrical outputs and to convert them into voltage form for processing by the ADC.

You should be aware of a number of important sensor characteristics in order to successfully design and write interface software. Of most relevance are accuracy, dynamic range, stability, linearity, susceptibility to noise, and response times. The latter includes rise time and settling time and is closely related to the sensor's frequency response.

Sensor characteristics cannot be considered in isolation. Sensors are often closely coupled to their signal-conditioning circuits and we must, therefore, also take into account the performance of this component when designing a DA&C system. Signal-conditioning and digitization circuitry can play an important (if not the most important) role in determining the characteristics of the measuring system as a whole. Although signal-conditioning circuits can introduce undesirable properties of their own, such as noise or drift, they are usually designed to compensate for inadequacies in the sensor's response. If properly matched, signal-conditioning circuits are often able to cancel out sensor offsets, non-linearities or temperature dependencies. We will discuss signal conditioning later in this chapter.

Accuracy

Accuracy represents the precision with which a sensor can respond to the measurand. It refers to the overall precision of the device resulting from the combined effect of offsets and proportional measurement errors. When assessing accuracy, one must take account of manufacturers' figures for repeatability, hysteresis, stability and, if appropriate, resolution. Although a sensor's accuracy figure may include the effect of resolution, the two terms must not be confused. Resolution represents the smallest *change* in the measurand that the sensor can detect. Accuracy includes this, but also encompasses other sources of error.

Dynamic range

A sensor's dynamic range is the ratio of its full-scale value to the minimum detectable signal variation. Some sensors have very wide

dynamic ranges and, if the full range is to be accommodated, it may be necessary to employ high resolution ADCs or Programmable-Gain Amplifiers (PGAs). Using a PGA might increase the system's data-storage requirements, because of the addition of an extra variable (i.e. gain). These topics are discussed further in the section *Amplification and extending dynamic range* later in this chapter.

Stability and repeatability

The output from some sensors tends to drift over time. Instabilities may be caused by changes in operating temperature or by other environmental factors. If the sensor is likely to exhibit any appreciable instability, you should assess how this can be compensated for in the software. You might wish, for example, to include routines which force the operator to recalibrate or simply rezero the sensor at periodic intervals (see Chapter 9). Stability might also be compromised by small drifts in the supplied excitation signals. If this is a possibility, the software should be designed to monitor the excitation voltage using a spare analogue input channel and to correct the measured sensor readings accordingly.

Linearity

Most sensors provide a linear output – i.e. their output is directly proportional to the value of the measurand. In such cases the sensor response curve consists of a straight line. Some devices such as thermocouples do not exhibit this desirable characteristic. If the sensor output is not linearized within the signal-conditioning circuitry, it will be necessary for the software to correct for any non-linearities present. Chapter 9 demonstrates several software linearization techniques.

Response times

The time taken by the sensor to respond to an applied stimulus is obviously an important limiting factor in determining the overall throughput of the system. The sensor's response time (sometimes expressed in terms of its frequency response) should be carefully considered, particularly in systems which monitor for dangerous, over-range or otherwise erroneous conditions. Many sensors provide a virtually instantaneous response and in these cases it is usually the signal-conditioning or digitization components (or, indeed, the software itself) which determines the maximum possible throughput. This is not generally the case with temperature sensors, however.

Semiconductor sensors, thermistors and thermocouples tend to exhibit long response times (upwards of 1 s). In these cases, there is little to be gained (other than the ability to average out noise) by sampling at intervals shorter than the sensor's time constant.

You should be careful when interpreting response times published in manufacturers' literature. They often relate to the time required for the sensor's output to change by a fixed fraction in response to an applied step change in temperature. If a time *constant* is specified it generally defines the time required for the output to change by $1 - e^{-1}$ (i.e. about 63.21 per cent) of the difference between its initial and final steady state outputs. The response time will be longer if quoted for a greater fractional change. The response time of thermal sensors will also be highly dependent upon their environment. Thermal time constants are usually quoted for still air, but much faster responses will apply if the sensor is immersed in a free-flowing or stirred liquid such as oil or water.

Susceptibility to noise

Noise is particularly problematic with sensors which generate only low level signals (e.g. thermocouples and strain gauges). Low-pass filters can be used to remove noise which often occurs predominantly at higher frequencies than the signals to be measured. Steps should always be taken to exclude noise at its source by adopting good shielding and grounding practices. As signal-conditioning circuits and cables can introduce noise themselves, it is essential that they are well designed. Even when using hardware and electronic filters, there may still be some residual noise on top of the measured signal. A number of filtering techniques can be employed in the software and some of these are discussed in Chapter 4.

Some common sensors

This section describes features of several common sensors which are relevant to DA&C software design. Unfortunately, space does not permit an exhaustive list. Many sensors that do not require special considerations or software techniques are excluded from this section. Some less widely used devices, such as optical and chemical sensors are also excluded, even though they are often associated with problems such as long response times and high noise levels. Details of the operation of these devices may be found in specialist books such as Tompkins and Webster (1988), Parr (1986) or Warring and Gibilisio (1985).

The information provided below is typical for each type of sensor described. However, different manufacturers' implementations vary considerably. The reader is advised to consult manufacturers' data sheets for precise details of the sensor and signal-conditioning circuits which they intend to use.

Digital sensors and encoders

Some types of sensor convert the analogue measurand into an equivalent digital representation which can be transferred directly to the PC. Digital sensors tend to require minimal signal conditioning.

As mentioned above the simplest form of digital sensor is the switch. Examples include inductive proximity switches and mechanical limit switches. These produce a single-bit input which changes state when some physical parameter (e.g. spatial separation or displacement) rises above, or falls below, a predefined limit. However, to measure the *magnitude* of an analogue quantity we need a sensor with a response which varies in many (typically several hundred of more) steps over its measuring range. Such sensors are more correctly known as encoders as they are designed to encode the measurand into a digital form.

Sensors such as the rotor tachometer employ magnetic pickups which produce a stream of digital pulses in response to the rotation of a ferrous disk. Angular velocity or incremental changes in angular position can be measured with these devices. The pulse rate is proportional to the angular velocity of the disk. Similar sensors are available for measuring linear motion.

Shaft encoders are used for rotary position or velocity measurement in a wide range of industrial applications. They consist of a binary encoded disk which is mounted on a rotating shaft or spindle and located between some form of optical transmitter and matched receiver (e.g. infrared LEDs and phototransistors). The bit pattern detected by the receiver will depend upon the angular position of the encoded disk. The resolution of the system might be typically $\pm 1°$.

A disk encoded in true (natural) binary has the potential to produce large errors. If, for example, the disk is very slightly misaligned, the most significant bit might change first during a transition between two adjacent encoded positions. Such a situation can give rise to a momentary 180° error in the output. This problem is circumvented by using the Gray code. This a binary coding scheme in which only one bit changes between adjacent coded positions. The outputs from these encoders are normally converted to digital pulse trains which carry rotary position, speed and direction information. Because of this it is rarely necessary for the DA&C programmer to

use binary Gray codes directly. We will, however, discuss other binary codes later in this chapter.

The signals generated by digital sensors are often not TTL compatible, and in these cases additional circuitry is required to interface to the PC. Some or all of this circuitry may be supplied with (or as part of) the sensor, although certain TTL buffering or opto-isolation circuits may have to be provided on separate plug-in digital interface cards.

Digital position encoders are inherently linear, stable and immune to electrical noise. However, care has to be taken when *absolute* position measurements are required, particularly when using devices which produce identical pulses in response to *incremental* changes in position. The measurement must always be accurately referenced to a known zero position. Systematic measurement errors can result if pulses are somehow missed or not counted by the software. Regular zeroing of such systems is advisable if they are to be used for repeated position measurements.

Potentiometric sensors

These very simple devices are usually used for measurement of linear or angular position. They consist of a resistive wire and sliding contact. The resistance to the current flowing through the wire and contact is a measure of the position of the contact. The linearity of the device is determined by the resistance of the output load, but with appropriate signal conditioning and buffering, non-linearities can generally be minimized and may, in fact, be negligible. Most potentiometric sensors are based on closely wound wire coils. The contact slides along the length of the coil and as it moves across adjacent windings it produces a stepped change in output. These steps may limit the resolution of the device to typically 25 to 50 μm.

Semiconductor temperature sensors

This class of temperature sensor includes devices based on discrete diodes and transistors as well as temperature-sensitive integrated circuits. Most of these devices are designed to exhibit a high degree of stability and linearity. Their working range is, however, relatively limited. Most operate from about −50 to +150°C, although some devices are suitable for use at temperatures down to about −230°C or lower. IC temperature sensors are typically linear to within a few degrees centigrade. A number of ICs and discrete transistor temperature sensors are somewhat more linear than this: perhaps ±0.5 to ±2°C or better. The repeatability of some devices may be as low as ±0.01°C.

All thermal sensors tend to have quite long response times. Their time constants are dependent upon the rate at which temperature changes are conducted from the surrounding medium. The intrinsic time constants of semiconductor sensors are usually of the order of 1–10 s. These figures assume efficient transmission of thermal energy to the sensor. If this is not the case, much longer time constants will apply (e.g. a few seconds to about one minute in still air).

Most semiconductor temperature sensors provide a high level current or voltage output which is relatively immune to noise and can be interfaced to the PC with minimal signal conditioning. Because of the long response times, software filtering can be easily applied should noise become problematic.

Thermocouples

Thermocouples are very simple temperature measuring devices. They consist of junctions of two dissimilar metal wires. An electromotive force (emf) is generated at each of the thermocouple's junctions by the Seebeck effect. The magnitude of the emf is directly related to the temperature of the junction. Various types of thermocouple are available for measuring temperatures from about −200°C to in excess of 1800°C. There are a number of considerations which must be borne in mind when writing interface software for thermocouple systems.

Depending upon the type of material from which the thermocouple is constructed, its output ranges from about 10 to 70 μV/°C. Thermocouple response characteristics are defined by various British and international standards. The sensitivity of thermocouples tends to change with temperature and this gives rise to a non-linear response. The non-linearity may not be problematic if measurements are to be confined to a narrow enough temperature range, but in most cases there is a need for some form of linearization. This may be handled by the signal conditioning circuits, but it is often more convenient to linearize the thermocouple's output by means of suitable software algorithms. Chapter 9 illustrates a number of linearization techniques which can be applied to thermocouples.

Even when adequately linearized, thermocouple-based temperature measuring systems are not awfully accurate, although it has to be said that they are often more than adequate for many temperature-sensing applications. Thermocouple accuracy is generally limited by variations in manufacturing processes or materials to about 1 to 4°C.

Like other forms of temperature sensor, thermocouples have long response times. This depends upon the mass and shape of the thermocouple and its sheath. According to the Labfacility temperature

sensing handbook (1987), time constants for thermocouples in still air range from 0.05 to around 40 s.

Thermocouples are rather insensitive devices. They output only low level signals – typically less than 50 mV – and are, therefore, prone to electrical noise. Unless the devices are properly shielded, mains pickup and other forms of noise can easily swamp small signals. However, because thermocouples respond slowly, their outputs are very amenable to filtering. Heavy software filtering can usually be applied without losing any important temperature information.

Cold-junction compensation

In order to form a complete circuit the conductors which make up the thermocouple must have at least two junctions. One (the sensing junction) is placed at an unknown temperature (i.e. the temperature to be measured) and the remaining junction (known as the cold junction or reference junction) is either held at a fixed reference temperature or allowed to vary (over a narrow range) with ambient temperature. The reference junction generates its own temperature-dependent emf which must be taken into account when interpreting the total measured thermocouple voltage.

Thermocouple outputs are usually tabulated in a form that assumes that the reference junction is held at a constant temperature of 0°C. If the temperature of the cold junction varies from this fixed reference value, the additional thermal emf will offset the sensor's response. It is not possible to calibrate out this offset unless the temperature of the cold junction is known and is constant. Instead, the cold junction's temperature is normally monitored in order that a dynamic correction may be applied to the measured thermocouple voltage.

The cold-junction temperature can be sensed using an independent device such as a semiconductor (transistor or IC) temperature sensor. In some signal-conditioning circuits, the output from the semiconductor sensor is used to generate a voltage equal in magnitude, but of opposite sign, to the thermal emf produced by the cold junction. This voltage is then electrically added to the thermocouple signal so as to cancel any offset introduced by the temperature of the cold junction.

It is also possible to perform a similar offset-cancelling operation within the data-acquisition software. If the output from the semiconductor temperature sensor is read via an ADC, the program can gauge the cold-junction temperature. As the thermocouple's response curve is known, the software is able to calculate the thermal emf produced by the cold junction – i.e. the offset value. This is then applied to the total measured voltage in order to determine that part

of the thermocouple output due only to the *sensing* junction. This is accomplished as follows.

The response of the cold junction and the sensing junction both generally follow the same non-linear form. As the temperature of the cold junction is usually limited to a relatively narrow range, it is often practicable to approximate the response of the cold junction by a straight line:

$$T_{CJ} = a_0 + a_1 V_{CJ} \tag{3.1}$$

where T_{CJ} is the temperature of the cold junction in °C, V_{CJ} is the corresponding thermal emf and a_0 and a_1 are constants which depend upon the thermocouple type and the temperature range over which the straight-line approximation is made. Table 3.1 lists the parameters of straight-line approximations to the response curves of a range of different thermocouples over the temperature range from 0 to 40°C.

The measured thermocouple voltage V_M is equal to the difference between the thermal emf produced by the sensing junction (V_{SJ}) and the cold junction (V_{CJ}):

$$V_M = V_{SJ} - V_{CJ} \tag{3.2}$$

As we are interested only in the difference in junction voltages, V_{SJ} and V_{CJ} can be considered to represent either the absolute thermal emfs produced by each junction or the emfs *relative* to whatever junction voltage might be generated at some convenient temperature origin. In the following discussion we will choose the origin of the temperature scale to be 0°C (so that 0°C is considered to produce a zero junction voltage). In fact, the straight-line parameters listed in Table 3.1 represent an approximation to a 0°C-based response curve (a_0 is close to zero).

Rearranging Equation 3.1 and substituting for V_{CJ} in Equation 3.2 we see that

$$V_{SJ} = V_M + \frac{T_{CJ} - a_0}{a_1} \tag{3.3}$$

The values of a_0 and a_1 for the appropriate type of thermocouple can be substituted from Table 3.1 into this equation in order to compensate for the temperature of the cold junction. All voltage values should be in millivolts and T_{CJ} should be expressed in °C. The temperature of the sensing junction can then be calculated by applying a suitable linearizing polynomial to the V_{SJ} value, as described in Chapter 9. Note that the polynomial must also be

Table 3.1 *Parameters of straight-line fits to thermocouple response curves over the range 0 to 40°C, for use in software cold-junction compensation*

Type	a_0*(°C)*	a_1*(°C mV^{-1})*	*Accuracy(°C)*
K	0.130	24.82	±0.25
J	0.116	19.43	±0.25
R	0.524	172.0	±1.00
S	0.487	170.2	±1.00
T	0.231	24.83	±0.50
E	0.174	16.53	±0.30
N	0.129	37.59	±0.40

constructed for a coordinate system with an origin at $V = 0$ mV, $T = 0$°C.

It is interesting to note that the type B thermocouple is not amenable to this method of cold-junction compensation as it exhibits an unusual behaviour at low temperatures. As the temperature rises from zero to about 21°C, the thermoelectric voltage falls to approximately -3 μV. It then begins to rise, through 0 V at about 41°C, and reaches $+3$ μV at 52°C. It is, therefore, not possible to accurately fit a straight line to the thermocouple's response curve over this range. Fortunately, if the cold-junction temperature remains within 0 to 52°C it contributes only a small proportion of the total measured voltage (less than about ±3 μV). If the sensing junction is used over its normal working range of 600 to 1700°C, the measurement error introduced by completely ignoring the cold junction emf will be less than ±0.6°C.

The accuracy figures quoted in Table 3.1 are generally better than typical thermocouple tolerances and so the a_0 and a_1 parameters should be usable in most situations. More precise compensation factors can be obtained by fitting the straight line over a narrower temperature range or by using a look-up table with the appropriate interpolation routines (see Chapter 9). You should calculate your own compensation factors if a different cold-junction temperature range is to be used.

Resistive temperature sensors (thermistors and RTDs)

Thermistors are semiconductor or metal oxide devices whose resistance changes with temperature. Most exhibit negative temperature coefficients (i.e. their resistance decreases with increasing temperature) although some have positive temperature coefficients.

Thermistor temperature coefficients range from about 1 to 5 per cent/°C. They tend to be usable in the range −70 to +150°C, but some devices can measure temperatures up to 300°C. Thermistor-based measuring systems can generally resolve temperature changes as small as ±0.01°C, although typical devices can provide absolute accuracies no better than ±0.1 to 0.5°C. The better accuracy figure is often only achievable in devices designed for use over a limited range (e.g. 0 to 100°C).

As shown in Figure 3.4, thermistors tend to exhibit a highly non-linear response. This can be corrected by means of suitable signal-conditioning circuits or by combining thermistors with positive and negative temperature coefficients. Although this technique can provide a high degree of linearity, it may be preferable to carry out linearization within the DA&C software. A third order *logarithmic* polynomial is usually appropriate (see Chapter 9). The response time of thermistors depends upon their size and construction. They tend to be comparable with semiconductor temperature sensors in this respect, but because of the range of possible constructions, thermistor time constants may be as low as several tens of milliseconds or as high as 100–200 s.

Resistance Temperature Detectors (RTDs) also exhibit a temperature-dependent resistance. These devices can be constructed from a variety of metals, but platinum is the most widely used. They are suitable for use over ranges of about −270 to 660°C, although some devices have been employed for temperatures up to about 1000°C. RTDs are accurate to within typically 0.2 to 4°C, depending

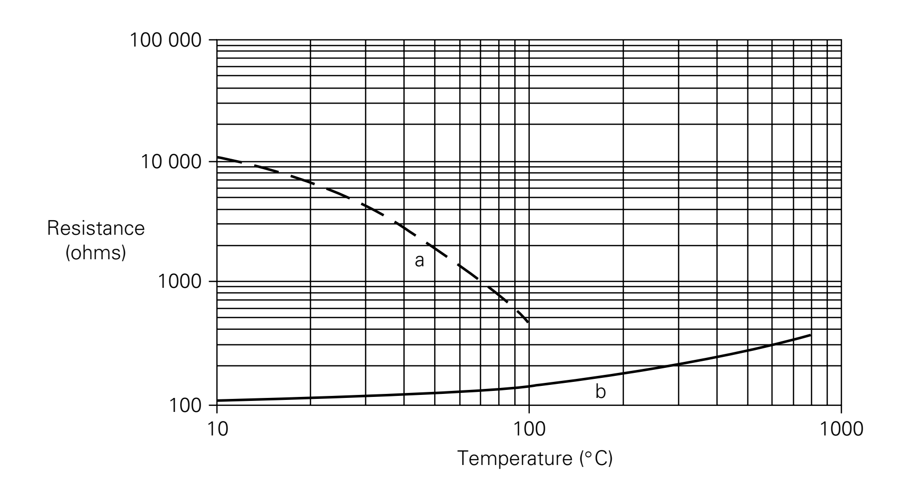


Figure 3.4 *Typical resistance vs. temperature characteristics for (a) negative temperature coefficient thermistors and (b) platinum RTDs*

on temperature and construction. They also exhibit a good long-term stability, so frequent recalibration may not be necessary. Their temperature coefficients are generally of the order of 0.4 Ω/°C. However, their sensitivity falls with increasing temperature, leading to a slightly non-linear response. This non-linearity is often small enough, over limited temperature ranges (e.g. 0 to 100°C), to allow a linear approximation to be used. Wider temperature ranges require some form of linearization to be applied: a third order polynomial correction usually provides the optimum accuracy. Response times are comparable with those of thermistors.

Resistance sensors and bridges

A number of other types of resistance sensor are available. Most notable amongst these are strain gauges. These take a variety of forms, including semiconductors, metal wires and metal foils. They are strained when subjected to a small displacement and, as the gauge becomes deformed, its resistance changes slightly. It is this resistance which is indirectly measured in order to infer values of strain, force or pressure. The Light Dependent Resistor (LDR) is another example of a resistance sensor. The resistance of this device changes in relation to the intensity of light impinging upon its surface.

Both thermistors and RTDs can be used in simple resistive networks, but, because devices such as RTDs and strain gauges have low sensitivities it can be difficult to directly measure changes in resistance. Bridge circuits such as that shown in Figure 3.5 are, therefore, often used to obtain optimum precision. The circuit is designed (or adjusted) so that the voltage output from the bridge is zero at some convenient value of the measurand (e.g. zero strain in the case of a strain gauge bridge). Any changes in resistance induced by changes in the measurand cause the bridge to become unbalanced and to produce a small output voltage. This can be amplified and measured independently of the much larger bridge-excitation voltage. Although bridge circuits are used primarily with insensitive devices, they can also be used with more responsive resistance sensors such as thermistors.

Bridges often contain two or four sensing elements (replacing the fixed resistors shown in Figure 3.5). These are arranged in such a way as to enhance the overall sensitivity of the bridge and, in the case of non-thermal sensors, to compensate for temperature dependencies of the individual sensing elements. This approach is used in the design of strain-gauge-based sensors such as load cells or pressure transducers.

Bridges with one sensing element exhibit a non-linear response. Two-active-arm bridges, which have sensors placed in opposite arms,

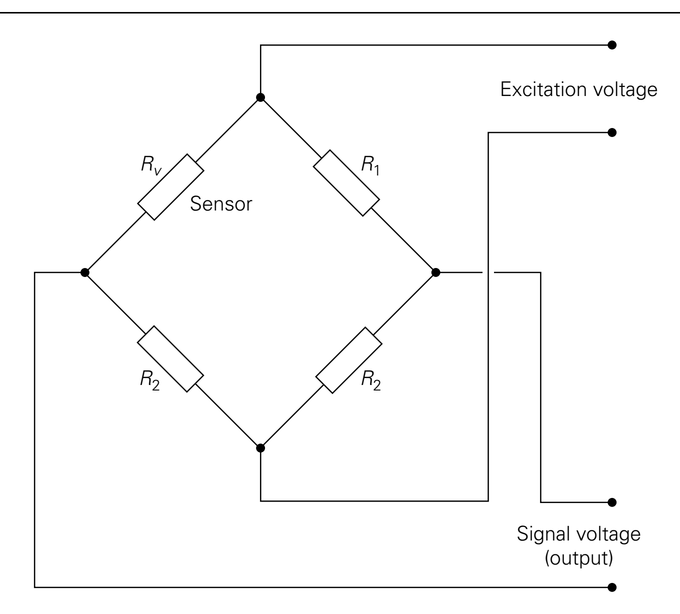


Figure 3.5 *Bridge circuit for measuring resistance changes in strain gauges and RTDs*

are also non-linear. However, provided that only small fractional changes occur in the resistance of the sensing element(s), the non-linearities of one and two arm bridges are often small enough that they can be ignored. Strain-gauge bridges with four active sensors generate a linear response provided that the sensors are arranged so that the resistance change occurring in one diagonally opposing pair of gauges is equal and opposite to that occurring in the other (Pople, 1979). When using resistance sensors in a bridge configuration, it is advisable to check for and, if necessary, correct any non-linearities that may be present. Linearization and calibration of strain-gauge bridges is discussed in Chapter 9.

Conduction of the excitation current can cause self-heating within each sensing element. This can be problematic with thermal sensors – thermistors in particular. Temperature rises within strain gauges can also cause errors in the bridge output. Because of this, excitation currents and voltages have to be kept within reasonable limits. This often results in low signal levels. For example, in most implementations, strain-gauge bridges generate outputs of the order of a few millivolts. Because of this, strain-gauge and RTD-based measuring systems are susceptible to noise, and a degree of software or hardware filtering is frequently required.

Lead resistance must also be considered when using resistance sensors. This is particularly so in the case of low resistance devices

such as strain gauges and RTDs, which have resistances of typically 120 to 350 Ω and 100 to 200 Ω, respectively. In these situations even the small resistance of the lead wires can introduce significant measurement errors. The effect of lead resistance can be minimized by means of compensating cables and suitable signal conditioning. This is usually the most efficient approach. Alternatively, the same type of compensation can be performed in software by using a spare ADC channel to directly measure the excitation voltage at the location of the sensor or bridge.

Linear variable differential transformers (LVDTs)

Linear Variable Differential Transformers (LVDTs) are used for measuring linear displacement. They consist of one primary and two secondary coils. The primary coil is excited with a high frequency (typically several hundred to several thousand Hz) voltage. The magnetic-flux linkage between the concentric primary and secondary coils depends upon the position of a ferrite core within the coil geometry. Induced signals in the secondary coils are combined in a differential manner such that movement of the core along the axis of the coils results in a variation in the amplitude and phase of the combined secondary-coil output. The output changes phase at the central (null) position and the amplitude of the output increases with displacement from the null point. The high frequency output is then demodulated and filtered in order to produce a DC voltage in proportion to the displacement of the ferrite core from its null position. The filter used is of the low-pass type which blocks the high frequency ripple but passes lower frequency variations due to core movement.

Obviously the excitation frequency must be high in order to allow the filter's cut-off frequency to be designed such that it does not adversely affect the response time of the sensing system. The excitation frequency should be considerably greater than the maximum frequency of core movement. This is usually the case with LVDTs. However, the filtration required with low frequency excitation (less than a few hundred Hz) may significantly affect the system's response time and must be taken into account by the software designer.

The LVDT offers a high sensitivity (typically 100–200 mV/V at its full-scale position) and high level voltage output which is relatively immune to noise. Software filtering can, however, enhance noise rejection in some situations.

The LVDT's intrinsic null position is very stable and forms an ideal reference point against which to position and calibrate the sensor. The resolution of an LVDT is theoretically infinite. In practice, however, it is limited by noise and the ability of the

signal-conditioning circuit to sense changes in the LVDT's output. Resolutions of less than 1 μm are possible. The device's repeatability is also theoretically infinite, but is limited in practice by thermal expansion and mechanical stability of the sensor's body and mountings. Typical repeatability figures lie between ±0.1 and ±10 μm, depending upon the working range of the device. Temperature coefficients are also an important consideration. These are usually of the order of 0.01 per cent/°C. It is wise to periodically recalibrate the sensor, particularly if it is subject to appreciable temperature variations.

LVDTs offer quite linear responses over their working range. Designs employing simple parallel coil geometries are capable of maintaining linearity over only a short distance from their null position. Non-linearities of up to 10 per cent or more become apparent if the device is used outside this range. In order to extend their operating range, LVDTs are usually designed with more complex and expensive graduated or stepped windings. These provide linearities of typically 0.25 per cent. An improved linearity can sometimes be achieved by applying software linearization techniques as described in Chapter 9.

3.4 Handling analogue signals

Signal levels and current-loading requirements of sensors and actuators usually preclude their direct connection to ADCs and DACs. For this reason, data-acquisition and control systems generally require analogue signals to be processed before being input to the PC, or after transmission from it. This usually involves conditioning (i.e. amplifying, filtering and buffering) the signal. In the case of analogue inputs it may also entail selecting and capturing the signal using devices such as multiplexers and sample-and-hold circuits.

Signal conditioning

Signal conditioning is normally required on both inputs and outputs. In this section we will concentrate on analogue inputs, but analogous considerations will apply to analogue outputs: for example, the circuits used to drive actuators.

Conditioning analogue inputs

Signal conditioning serves a number of purposes. It is needed to clean and shape signals, to supply excitation voltages, to amplify and buffer low level signals, to linearize sensor outputs, to compensate for temperature-induced drifts and to protect the PC from electrical

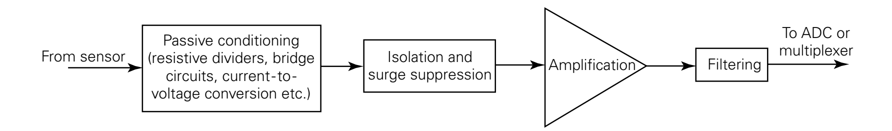


Figure 3.6 *Elements of a typical analogue input signal-conditioning circuit*

noise and surges. The signal-conditioning blocks shown in Figure 3.2 may consist of a number of separate circuits and components. These elements are illustrated in Figure 3.6.

Certain passive signal-conditioning elements such as potential dividers, bridge circuits and current-to-voltage conversion resistors are often closely coupled to the sensor itself and, indeed, may be an integral part of it. The sensor is sometimes isolated from the remaining signal-conditioning circuits and from the PC by means of linear opto-couplers or capacitively coupled devices. Surge-suppression components such as Zener diodes and metal oxide varistors may also be used in conjunction with RC networks to protect against transient voltage spikes.

Because typical ADCs have sensitivities of a few millivolts per bit, it is essential to amplify the low level signals from thermocouples, strain gauges and RTDs (which may be only a few tens of millivolts at full scale). Depending upon the type of sensor in use, activities such as AC demodulation or thermocouple cold-junction compensation might also be performed prior to amplification. Finally, a filtering stage might be employed to remove random noise or AC excitation ripple. Low-pass filters also serve an anti-aliasing function as described in Chapter 4.

So what relevance does all this have to the DA&C programmer? In well-designed systems, very little – the characteristics of the signal conditioning *should* have no significant limiting affect on the design or performance of the software, and most of the characteristics of the sensor and signal conditioning *should* be transparent to the programmer. Unfortunately this is not always the case.

The amplifier and other circuits can give rise to temperature-dependent offsets or gain drifts (typically of the order of 0.002–0.010 per cent of full scale per °C) which may necessitate periodic recalibration or linearization. When designing DA&C software you should consider the following:

- the frequency of calibration
- the need to enforce calibration or to prompt the operator when calibration is due
- how calibration data will be input, stored and archived
- the necessity to rezero sensors after each data-acquisition cycle.

You should also consider the frequency response (or bandwidth) of the signal-conditioning circuitry. This can affect the sampling rate and limit throughput in some applications (see Chapter 4). Typical bandwidths are of the order of a few hundred Hz, but this does, of course, vary considerably between different types of signal-conditioning circuit and depends upon the degree of filtration used. High gain signal-conditioning circuits, which amplify noisy low level signals, often require heavy filtering. This may limit the bandwidth to typically 100 to 200 Hz. Systems employing low frequency LVDTs can have even lower bandwidths. Bandwidth may not be an important consideration when monitoring slowly varying signals (e.g. temperature), but it can prove to be problematic in high speed applications involving, for example, dynamic force or strain measurement.

If high gain amplifiers are used and/or if hardware filtration is inadequate, it may be necessary to incorporate filtering algorithms within the software. If this is the case, you should carefully assess which signal frequencies you wish to remove and which frequencies you will need to retain, and then reconcile this with the proposed sampling rate and the software's ability to reconstruct an accurate representation of the underlying noise-free signal. Sampling considerations and software filtering techniques are discussed in Chapter 4.

It may also, in some situations, be necessary for the software to monitor voltages at various points within the signal-conditioning circuit. We have already mentioned monitoring of bridge excitation levels to compensate for voltage drops due to lead-wire resistance. The same technique (sometimes known as ratiometric correction) can also be used to counteract small drifts in excitation supply. If lead-wire resistance can be ignored, the excitation voltage may be monitored either at its source or at the location of the sensor.

There is another (although rarer) instance when it might be necessary to monitor signal-conditioning voltage levels. This is when pseudo-differential connections are employed on the input to an amplifier. Analogue signal connections may be made in two ways: single ended or differential. Single-ended signals share a common ground or return line. Both the signal source voltage and the input to the amplifier(s) exist relative to the common ground. For this method to work successfully, the ground potential difference between the source and amplifier must be negligible otherwise the signal to be measured appears superimposed on a non-zero (and possibly noisy) ground voltage. If a significant potential difference exists between the ground connections, currents can flow along the ground wire causing errors in the measured signals.

Differential systems circumvent this problem by employing two wires for each signal. In this case, the signal is represented by the potential difference between the wires. Any ground-loop-induced voltage appears equally (as a common-mode signal) on each wire and can be easily rejected by a differential amplifier.

An alternative to using a full differential system is to employ pseudo-differential connections. This scheme is suitable for applications in which the common-mode voltage is moderately small. It makes use of single-ended channels with a common ground connection. This allows cheaper operational amplifiers to be used. The potential of the common ground return point is measured using a spare ADC input in order to allow the software to correct for any differences between the local and remote ground voltages. Successful implementation of this technique obviously requires the programmer to have a reasonably detailed knowledge of the signal conditioning circuitry. Unless the common-mode voltage is relatively static, this technique also necessitates concurrent sampling of the signal and ground voltages. In this case simultaneous sample-and-hold circuits (discussed later in this chapter) or multiple ADCs may have to be used.

Conditioning analogue outputs

Some form of signal conditioning is required on most analogue outputs, particularly those that are intended to control motors and other types of actuator. Space limitations preclude a detailed discussion of this topic, but in general, the conditioning circuits include current-driving devices and power amplifiers etc. The nature of the signal conditioning used is closely related to the type of actuator. As in the case of analogue inputs, it is prudent for the programmer to gain a thorough understanding of the actuator and associated signal-conditioning circuits in order that the software can be designed to take account of any non-linearities or instabilities which might be present.

Multiplexers

Multiplexers allow several analogue input channels to be serviced by a single ADC. They are basically software-controlled analogue switches which can route one of typically 8 or 16 analogue signals through to the input of the system's ADC. A four-channel multiplexed system is illustrated in Figure 3.2. A multiplexer used in conjunction with a single ADC (and possibly amplifier) can take the place of several ADCs (and amplifiers) operating in parallel. This is normally considerably cheaper, and uses less power, than an

array of separate ADCs and for this reason analogue multiplexers are commonly used in multi-channel data-acquisition systems.

However, some systems do employ parallel ADCs in order to maximize throughput. The ADCs must, of course, be well matched in terms of their offset, gain and integral non-linearity errors. In such systems, the *digitized* readings from each channel (i.e. ADC) are *digitally* multiplexed into a data register or into one of the PC's I/O ports. From the point of view of software design, there is little to be said about digital multiplexers. In this section, we will deal only with the properties of their analogue counterparts.

In an analogue multiplexed system, multiple channels share the same ADC and the associated sensors must be read sequentially, rather than in parallel. This leads to a reduction in the number of channels that can be read per second. The decrease in throughput obviously depends upon how efficiently the software controls the digitization and data-input sequence.

A related problem is skewing of the acquired data. Unless special S/H circuitry is used, simultaneous sampling is not possible. This is an obvious disadvantage in applications which must determine the temporal relationship or relative phase of two or more inputs.

Multiplexers can be operated in a variety of ways. The desired analogue channel is usually selected by presenting a 3- or 4-bit address (i.e. channel number) to its control pins. In the case of a plug-in ADC card, the address-control lines are manipulated from within the software by writing an equivalent bit pattern to one of the card's registers (which usually appear in the PC's I/O space). Some systems can be configured to automatically scan a range of channels. This is often accomplished by programming the start and end channel numbers into a 'scan register'. In contrast, some intelligent DA&C units require a high-level channel-selection command to be issued. This often takes the form of an ASCII character string transmitted via a serial or parallel port.

Whenever the multiplexer is switched between channels, the input to the ADC or S/H will take a finite time to settle. The settling time tends to be longer if the multiplexer's output is amplified before being passed to the S/H or ADC. An instrumentation amplifier may take typically 1–10 μs to settle to a 12-bit (0.025 per cent) accuracy. The exact settling time will vary, but will generally be longest with high gain PGAs, or where the amplifier is required to settle to a greater degree of accuracy.

The settling time can be problematic. If the software scans the analogue channels (i.e. switches the multiplexer) too rapidly, the input to the S/H or ADC will not settle sufficiently and a degree of apparent cross-coupling may then be observed between adjacent

channels. This can lead to measurement errors of several per cent, depending upon the scanning rate and the characteristics of the multiplexer and amplifier used. These problems can be avoided by careful selection of components in relation to the proposed sampling rate. Bear in mind that the effects of cross-coupling may be dependent upon the sequence as well as the frequency with which the input channels are scanned. Cross-coupling may not even be apparent during some operations. A calibration facility, in which only one channel is monitored, will not exhibit any cross-coupling, while a multi-channel scanning sequence may be badly affected. It is advisable to check for this problem at an early stage of software development as, if present, it can impose severe restrictions on the performance of the system.

Sample-and-hold circuits

Many systems employ a sample-and-hold (S/H) circuit on the input to the ADC to freeze the signal while the ADC digitizes it. This prevents errors due to changes in the signal during the digitization process (see Chapter 4). In some implementations, the multiplexer can be switched to the next channel in a sequence as soon as the signal has been grabbed by the S/H. This allows the digitization process to proceed in parallel with the settling time of the multiplexer and amplifier, thereby enhancing throughput. S/H circuits can also be used to capture transient signals. Software-controlled systems are not capable of responding to very high speed transient signals (i.e. those lasting less than a few microseconds) and so in these cases, the S/H and digitization process may be *initiated* by means of special hardware (e.g. a pacing clock). The software is then notified (by means of an interrupt, for example) when the digitization process is complete.

S/H circuits require only a single digital control signal to switch them between their ‘sample’ and ‘hold’ modes. The signal may be manipulated by software via a control register mapped to one of the PC’s I/O ports, or it may be driven by dedicated on-board hardware. S/H circuits present at the input to ADCs are often considered to be an integral part of the digitization circuitry. Indeed, the command to start the analogue-to-digital conversion process may also automatically activate the S/H for the required length of time.

Simultaneous S/H

In multiplexed systems like that represented in Figure 3.2, analogue input channels have to be read sequentially. This introduces a time lag between the samples obtained from

successive channels. Assuming typical times for ADC conversion and multiplexer/amplifier settling, this time lag can vary from several tens to several hundreds of microseconds. The consequent skewing of the sample matrix can be problematic if you wish to measure the phase relationship between dynamically varying signals. Simultaneous S/H circuits are often used to overcome this problem. Figure 3.7 illustrates a four-channel analogue input system employing simultaneous S/H.

The system is still multiplexed, so very little improvement is gained in the overall throughput (total number of channels read per second), but the S/H circuits allow data to be captured from all inputs within a very narrow time interval (see the following section). Simultaneous S/H circuits may be an integral part of the signal conditioning unit or they may be incorporated in the digitization circuitry (e.g. on a plug-in ADC card). In either case they tend to be manipulated by a single digital signal generated by the PC.

Characteristics of S/H circuits

When not in use, the S/H circuit can be maintained in either the sample or hold modes. To operate the device, it must first be switched into sample mode for a short period and then into hold mode in order to freeze the signal before analogue-to-digital conversion begins. When switched to sample mode, the output of the S/H takes a short, but sometimes significant, time to react to its input. This time delay arises because the device has to charge up an internal capacitor to the level of the input signal. The rate of charging follows an exponential form and so a greater degree of accuracy is achieved if the capacitor is allowed to charge for a longer time. This charging time is known as the acquisition time. It varies considerably between different types of S/H circuit and, of course,

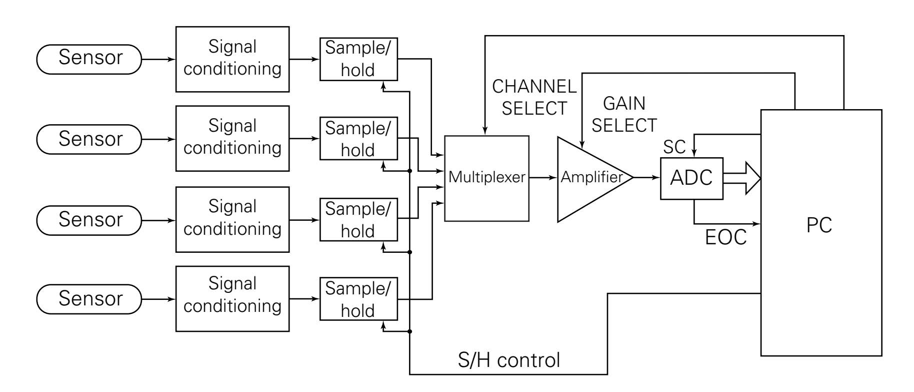


Figure 3.7 *Analogue input channels with simultaneous sample and hold*

depends upon the size of the voltage swing at the S/H's input. The worst case acquisition time is usually quoted and this is generally of the order of 0.5–20 μs. Acquisition time is illustrated, together with other S/H characteristics, in Figure 3.8. Accuracies of 0.01 per cent are often attainable with acquisition times greater than about 10 μs. Lower accuracies (e.g. 0.1 per cent) are typical of S/H devices working with shorter acquisition times.

While in sample mode, the S/H's output follows its input (provided that the hold capacitor has been accurately charged and that the signal does not change too quickly). When required, the device is switched into hold mode. A short delay then ensues before digitization can commence. The delay is actually composed of two constituent delay times known as the aperture time and the settling time. The former, which is due to the internal switching time of the device, is very short: typically less than 50 ns. Variations in the aperture time, known as aperture jitter (or aperture uncertainty time), are the limiting factor in determining the temporal precision of each sample. These variations are generally of the order of 1 ns, so aperture jitter can be ignored in all but the highest speed applications (see Chapter 4 for more on the relationship between aperture jitter and maximum sampling rate). The settling time is the time required for the output to stabilize after the switch and determines the rate at which samples can be obtained. It is usually of the order of 1 μs, but some systems exhibit much longer or shorter settling times.

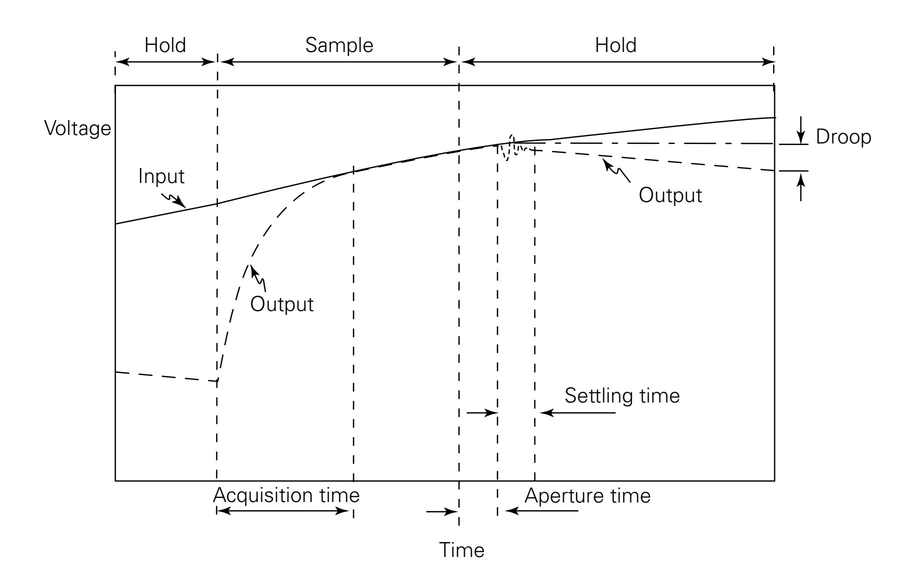


Figure 3.8 *Idealized sample-and-hold circuit response characteristic*

When the output settles to a stable state, it can be digitized by the ADC. Digitization must be completed within a reasonably short time interval because the charge on the hold capacitor begins to decay, causing the S/H's output to 'droop'. Droop rates vary between different devices, but are typically of the order of 1 mV/ms. Devices are available with both higher and lower droop rates. S/H circuits with low droop rates are usually required in simultaneous sample-and-hold systems. Large hold capacitors are needed to minimize droop and these can adversely affect the device's acquisition time.

3.5 Digitization and signal conversion

The PC is capable of reading and writing only digital signals. To permit interfacing of the PC to external analogue systems, ADCs and DACs must be used to convert signals from analogue to digital form and vice versa. This section describes the basic principles of the conversion processes. It also illustrates some of the characteristics of ADCs and DACs which you should be aware of when writing interface software.

Binary coding

In order to understand the digitization process, it is important to consider the ways in which analogue signals can be represented digitally. Computers store numbers in binary form. There are several binary coding schemes. Most positive integers, for example, are represented in true binary (sometimes called natural or straight binary). Just as the digits in a decimal number represent units, tens, hundreds etc., true binary digits represent 1s, 2s, 4s, 8s and so on. Floating-point numbers, on the other hand, are represented within the computer in a variety of different binary forms. Certain fields within the floating-point bit pattern are set aside for exponents or to represent the sign of the number. Although floating-point representations are needed to scale, linearize and otherwise manipulate data within the PC, all digitized analogue data are generally transferred in and out of the computer in the form of binary integers.

Analogue signals may be either unipolar or bipolar. Unipolar signals range from zero up to some positive upper limit, while bipolar signals can span zero, varying between non-zero negative and positive limits.

Encoding unipolar signals

Unipolar signals are perhaps the most common and are the simplest to represent in binary form. They are generally coded as true binary

numbers with which most readers should already be familiar. As mentioned above the least significant bit (LSB) has a weight (value) of 1 in this scheme, and the weight of each successive bit doubles as we move towards the most significant bit (MSB). If we allocate an index number, i, to each bit, starting with 0 for the LSB, the weight of any one bit is given by 2^i. Bit 6 would, for example, represent the value 2^6(=64 decimal). To calculate the value represented by a complete binary number, the weights of all non-zero bits must be added. For example, the following 8-bit true binary number would be evaluated as shown.

$$\begin{aligned} 1\,1\,0\,0\,1\,0\,0\,1 \text{ binary} &= 2^7 + 2^6 + 2^3 + 2^0 \\ &= 128 + 64 + 8 + 1 = 201 \text{ decimal} \end{aligned}$$

The maximum value which can be represented by a true binary number has all bits set to 1. Thus, a true binary number with n bits can represent values from 0 to V, where:

$$V = \sum_{i=0}^{i=n-1} 2^i = 2^n - 1 \tag{3.4}$$

An 8-bit true binary number can, therefore, represent integers in the range 0 to 255 decimal ($=2^8 - 1$). A greater range can be represented by binary numbers having more bits. Similar calculations for other numbers of bits yield the results shown in Table 3.2. The accuracies with which each true binary number can represent an analogue quantity are also shown.

The entries in this table correspond to the numbers of bits employed by typical ADCs and DACs. It should be apparent that converters with a higher resolution (number of bits) provide the potential for a greater degree of conversion accuracy.

When true binary numbers are used to represent an analogue quantity, the range of that quantity should be matched to the range

Table 3.2 *Ranges of true binary numbers*

Number of bits	*Range (true binary)*	*Accuracy (%)*
6	0 to 63	1.56
8	0 to 255	0.39
10	0 to 1 023	0.098
12	0 to 4 095	0.024
14	0 to 16 383	0.0061
16	0 to 65 535	0.0015

(i.e. V) of the ADC or DAC. This is generally accomplished by choosing a signal-conditioning gain which allows the full-scale range of a sensor to be matched exactly to the measurement range of the ADC. A similar consideration applies to the range of DAC outputs required to drive actuators. Assuming a perfect match (and that there are no digitizing errors), the limiting accuracy of any ADC or DAC system depends upon the number of bits available. An n-bit system can represent some physical quantity which varies over a range 0 to R, to a fractional accuracy $\pm\frac{1}{2}\delta R$ where:

$$\delta R = \frac{R}{2^n} \tag{3.5}$$

This is equal to the value represented by one LSB. True binary numbers are important in this respect as they are the basis for measuring the resolution of an ADC or DAC.

Encoding bipolar signals

Many analogue signals can take on a range of positive and negative values. It is, therefore, essential to be able to represent readings on both sides of zero as digitized binary numbers. Several different binary coding schemes can be used for this purpose. One of the most convenient and widely used is offset binary. As its name suggests, this scheme employs a true binary coding, which is simply offset from zero. This is best illustrated by an example. Consider a system in which a unipolar 0–10 V signal is represented in 12-bit true binary by the range of values from 0 to 4095. We can also represent a bipolar signal in the range −5 V to +5 V by using the same scaling factor (i.e. volts per bit) and simply shifting the zero-volt point halfway along the binary scale to 2048. An offset binary value of zero would, in this case, be equivalent to −5 V, and a value of 4095 would represent +5 V. Offset binary codes can, of course, be used with any number of bits.

Two's complement binary can also represent both positive and negative numbers. It employs a sign bit at the MSB location. This bit is 0 for positive numbers and 1 for negative numbers. Because one bit is dedicated to storing sign information, it cannot be used for coding the absolute magnitude of the binary number and so the range of *magnitudes* which can be represented by two's complement numbers is half that which can be accommodated by the same number of bits in true binary. To negate a positive binary integer, it is only necessary to complement (convert 0s to 1s and 1s to 0s) each bit and then add 1 to the result. Carrying out this operation – which is equivalent to multiplying by minus one – twice in succession yields the original number. As most readers will be aware, this scheme is

used by the IBM PC's 80x86 processor for storing and manipulating signed integers because it greatly simplifies the operations required to perform subtractive arithmetic. A number of ADCs, particularly those designed for audio and digital signal processing applications, also use this coding scheme.

There are a variety of less widely used methods of coding bipolar signals. For example, a simple true binary number, indicating magnitude, may be combined with an additional bit to record the sign of the number. Another encoding scheme is one's complement (or complementary straight) binary in which negative numbers are formed by simply inverting each bit of the equivalent positive true-binary number. Combinations of these coding schemes are sometimes used. For example, complementary offset binary consists of an offset binary scale in which each code is complemented. The result is that the zero binary code (all 0s) corresponds to the positive full-scale position, while the maximum binary code (all 1s) represents the negative full-scale position. Yet another scheme, complementary two's complement, is formed by simply inverting each bit of a two's complement value. These methods of binary coding are less important in PC applications although some ADCs may generate signed true binary or one's complement binary codes. Some DAC devices use the complementary offset binary scheme.

The various bipolar codes are compared in Table 3.3. This shows how a 3-bit binary number can represent values from −4 to +4 using the different coding schemes. The patterns shown in this table can be easily extended to numbers encoded using a greater number of bits. Note that only offset binary, complementary offset binary and two's complement binary have a unique zero code. Note also that these schemes are asymmetric about their zero point. Compare in particular the two forms of offset binary.

Table 3.3 *Comparison of bipolar binary codes*

Value	*Offset binary*	*Two's complement*	*One's complement*	*Complementary offset binary*
+3	111	011	011	000
+2	110	010	010	001
+1	101	001	001	010
0	100	000	000 or 111	011
−1	011	111	110	100
−2	010	110	101	101
−3	001	101	100	110
−4	000	100	–	111

Conversion from offset binary to two's complement binary is simply a matter of complementing the MSB. Complementing it again reverts back to offset binary encoding. It is a very straightforward task to convert between the various bipolar codes and examples will not be given here.

Other binary codes and related notations

There are two other binary codes which can be used in special circumstances: the Gray code and BCD. Both of these are, in fact, unipolar codes and cannot represent negative numbers without the addition of an extra sign bit. We have already introduced the Gray code in relation to digital encoders earlier in this chapter, but because the DA&C programmer rarely needs to use this code directly it will not be discussed further.

Binary coded decimal (BCD)

BCD is simply a means of encoding individual decimal digits in binary form. Each decimal digit is coded by a group of 4 bits. Although each group would be capable of recording 16 true binary values, only the lower 10 values (i.e. corresponding to 0 to 9, decimal) are used. The remaining values are unused and are invalid in BCD. A number with N decimal digits would occupy $4N$ bits, arranged such that the least significant group of 4 bits would represent the least significant decimal digit. For example:

1234 decimal = 0001 0010 0011 0100 BCD

ADCs which generate BCD output are used mostly for interfacing to decimal display devices such as panel meters. Most ADCs employed in PC applications (e.g. those on plug-in DA&C cards) use one of the coding schemes described previously, such as offset binary. However, a few components of the PC do make use of BCD. For example, the 16-bit 8254 timer counter used on AT compatible machines and on some plug-in data-acquisition cards can operate in a 4-decade BCD mode.

Hexadecimal notation

This is not a binary code. It is, in fact, a base-16 (rather than base-2) numeric representation. Hexadecimal notation is rather like BCD in that 4 bits are required for each hexadecimal digit. However, all 16 binary codes are valid and so each hexadecimal digit can represent the numbers from 0 to 15 (decimal). Hexadecimal numbers are written using an alphanumeric notation in which the lowest 10 digits

are represented by 0 to 9 and the remaining digits are written using the letters A to F. 'A' corresponds to 10 decimal, 'B' to 11 and so on. Hexadecimal numbers are followed by an 'h' to avoid confusing them with decimal numbers. The following example shows the binary and decimal equivalents of a 2-digit hexadecimal number:

$$3Ah = 0011\ 1010 \text{ binary} = (3 \times 16) + (10 \times 1) = 58 \text{ decimal}$$

Most numbers manipulated by computer software are coded using multiples of 4 bits: usually either 8, 16 or 32 bits. Hexadecimal is, therefore, a convenient shorthand method for expressing binary numbers and is used extensively in this and other publications.

Digital-to-analogue converters

Digital-to-analogue converters (DACs) have a variety of uses within PC-based DA&C systems. They may be used for waveform synthesis, to control the speed of DC motors, or to drive analogue chart recorders and meters. Many closed-loop control systems require analogue feedback from the PC and this is invariably provided by a DAC.

Most DACs generate full-scale outputs of a few volts (typically 0–10 V, ±5 V, or ±10 V). They have a limited current-driving capability (usually less than about 1–10 mA) and are often buffered using operational amplifiers. In cases where a low impedance or high power unit is to be driven, suitable power amplifiers may be required. Current-loop DACs with full-scale outputs of 4–20 mA are also available and these are particularly suited to long-distance transmission in noisy environments. Both bipolar and unipolar configurations are possible on many proprietary DAC cards by adjusting jumpers or DIP switches.

The resolution of a DAC is an important consideration. This is the number of input bits which the DAC can accept. As Equation 3.5 shows, it determines the accuracy with which the device can reconstruct analogue signals (also see Chapter 4). 8-bit and 12-bit DACs are, perhaps, the most common in DA&C applications although devices with a variety of other resolutions are available. Figure 3.9 shows the ideal transfer characteristic of a DAC. For reasons of clarity, this illustration is based on a hypothetical 3-bit DAC, having eight possible codes from 000b to 111b. Note that although there are eight codes, the DAC can only generate an output accurate to one-seventh of its maximum output voltage, which is one LSB short of its nominal full scale value, V_{max}.

DACs are generally controlled via registers mapped to one or more of the PC's I/O ports. When the desired bit pattern is written to the

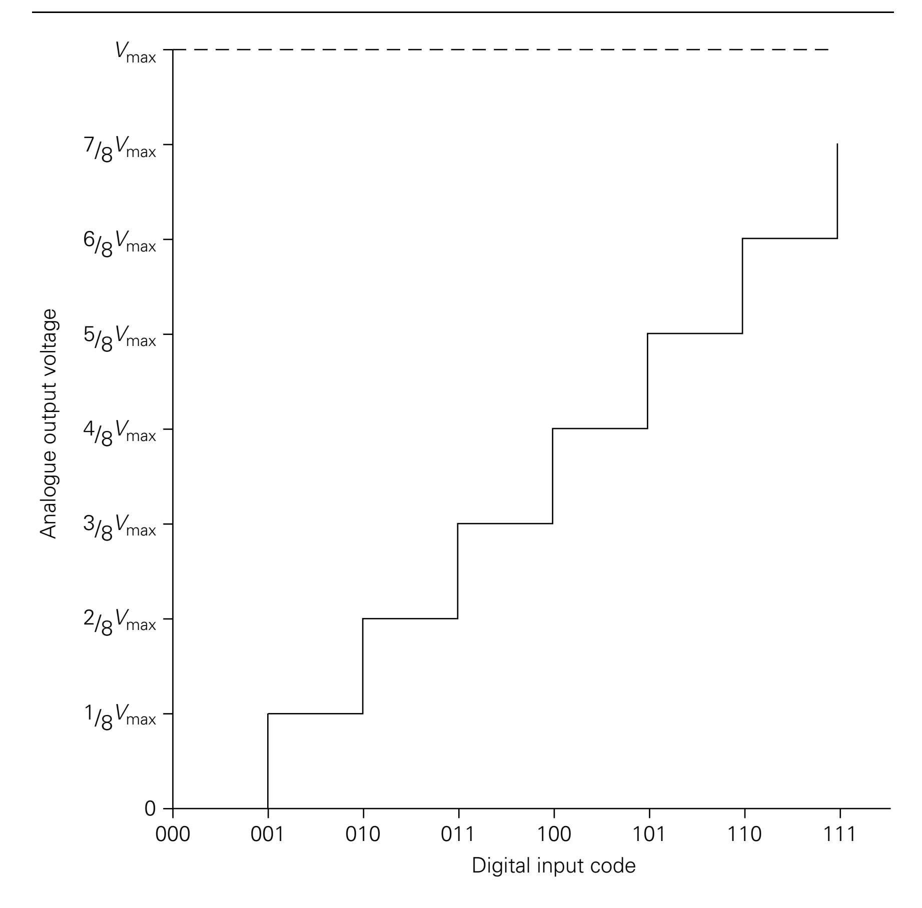


Figure 3.9 *Ideal DAC transfer characteristic (unipolar true binary encoding)*

register, the DAC updates its analogue output accordingly. If a DAC has more than 8 bits, it requires its digital input to be supplied either as one 16-bit word or as two 8-bit bytes. The latter often involves a two-stage write operation: the least significant byte is usually written first and this is followed by the most significant byte. Any unused bits (e.g. the upper 4 bits in the case of a 12-bit DAC) are ignored. The two-stage method of supplying new data can sometimes cause problems if the DAC's output is updated immediately upon receipt of each byte. Spurious transients can be generated because the least significant byte of the new data is initially combined with the most significant byte of the *existing* data. The analogue output settles to its desired value only when both new bytes have been supplied. To circumvent this problem, many DACs incorporate a double buffering system in which the first byte is held in a buffer until the second byte

is received, at which point the complete control word is transferred to the DAC's signal-generating circuitry.

Most devices employ a network of resistors and electronic switches connected to the input of an operational amplifier. The network is arranged such that each switch and its associated resistors make a binary-weighted contribution to the output of the amplifier. Each bit of the digital input operates one of the switches and thereby controls the input to, and output from, the amplifier. The operational amplifier and resistor network function basically as a multiplier circuit. It multiplies the digital input (expressed as a fraction of the full-scale digital input) by a fixed reference voltage. The reference voltage may be supplied by components external to the DAC. Most plug-in DA&C cards for the PC include suitable precision voltage references. Some also provide the facility for users to connect their own reference voltage and thereby to adjust the full-scale range of the DAC. Further details of DAC operation may be found in the texts by Tompkins and Webster (1988) and Vears (1990).

The output of a DAC can usually be updated quite rapidly. Each bit transition gives rise to transient fluctuations which require a short time to settle. The total settling time depends upon the number of bits that change during the update and is greatest when all input bits change (i.e. for a full-scale swing). The settling time may be defined as the time required after a full-scale input step for the DAC's output to settle to within a negligibly small band about its final level. The term 'negligibly small' has to be defined. Some DAC manufacturers define it as 'within $\pm\frac{1}{2}$ LSB', while others define it as a percentage of full scale, such as ±0.001 per cent. Quoted settling times range from about 0.1 to 150 μs, and sometimes up to about 1 ms, depending upon the characteristics of the device and on how the settling time is defined. Most DACs, however, have settling times of the order of 5–30 μs. In practice the overall settling time of an analogue output channel may be affected by external power amplifiers and other components connected to the DAC's outputs. You are advised to consult manufacturers' literature for precise timing specifications.

Characteristics of DACs

Because of small mismatches in components (e.g. the resistor network), it is not generally possible to fabricate DACs with the ideal transfer characteristic illustrated in Figure 3.9. Most DACs deviate slightly from the ideal, exhibiting several types of imperfection as shown in Figure 3.10. You should be aware of these potential sources of error in DAC outputs, some of which can be corrected by the use of appropriate software techniques.

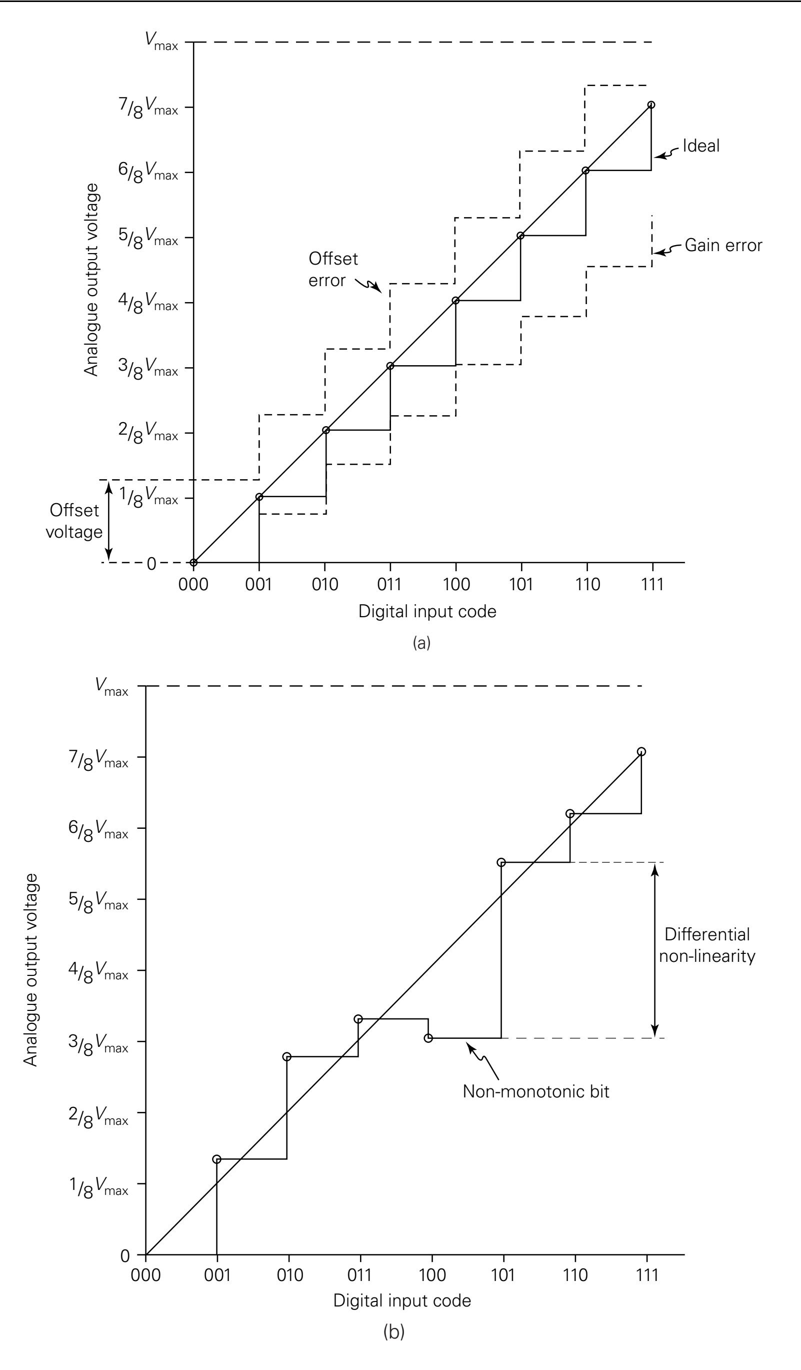


Figure 3.10 *Non-ideal DAC transfer characteristics: (a) gain and offset errors and (b) non-linearity and non-monotonicity*

The transfer characteristic may be translated along the analogue-output axis giving rise to a small offset voltage. Incorrect gains will modify the slope of the transfer characteristic such that the desired full-scale output is either obtained with a binary code lower than the ideal full-scale code (all 1s), or never reached at all. Gain errors equivalent to a few LSB are typical.

Linearity is a measure of how closely the output conforms to a straight line drawn between the end points of the conversion range. Linearity errors, which are due to small mismatches in the resistor network, cause the output obtained with some binary codes to deviate from the ideal straight-line characteristic. Most modern monolithic DACs are linear to within ±1 LSB or less. Differential non-linearity is the maximum change in analogue output occurring between any two adjacent input codes. It is defined in terms of the variation from the ideal step size of 1 LSB. Differential non-linearities are usually of the order of ±1 LSB or less. If non-linearity is such that the output from the DAC fails to increase over any single step in its input, the DAC is said to be non-monotonic. Monotonicity of a DAC is usually expressed as the number of bits over which monotonicity is maintained. If a DAC has a non-linearity better than $\pm\frac{1}{2}$ LSB, then it must be monotonic (it cannot be non-monotonic, by definition).

Although one can often compensate for gain and offset errors by manual trimming, it is not possible to correct non-linear or non-monotonic DACs – these characteristics are intrinsic properties of the device. Fortunately, most modern DAC designs yield quite small non-linearities which can usually be ignored. If, however, you are using a particularly non-linear device, you may wish to consider employing one of the linearization techniques described in Chapter 9.

Analogue-to-digital converters

An analogue-to-digital converter (ADC) is required to convert analogue sensor signals into a binary form suitable for reading into the PC. A wide variety of ADCs are available for this platform, either on plug-in DA&C cards or within remote signal-conditioning units or data loggers. This section introduces the basic concepts involved in analogue-to-digital conversion and describes some of the properties of ADCs which are relevant to the design of DA&C software.

Resolution and quantization error

It should be apparent to the reader that, because of the discrete nature of digital signals, some analogue information is lost in the

conversion process. A small but finite range of analogue input values are capable of generating any one digital output code. This range is known as the code width or, more properly, as a quantum as it represents the smallest change in analogue input which can be represented by the system. Its size corresponds to 1 LSB. The uncertainty introduced as a result of rounding to the nearest binary code is known as quantization error and has a magnitude equal to $\pm\frac{1}{2}$ LSB. Obviously, the quantization error is less important relative to the full-scale input range in ADCs that are capable of generating a wider range of output codes (i.e. those with a greater number of bits).

Some devices have a relatively low resolution of 8 bits or less, while others, designed for more precise measurements, may have 12 or 16 bits. ADCs usually have full-scale input ranges of a few volts: typically 0–10 V (unipolar) or ±5 V (bipolar). The quantization error is thus of the order of a few millivolts. Precise figures can easily be calculated by applying Equation 3.5, knowing the device's input range and resolution, as shown in the following example.

Consider a 12-bit ADC system designed for monitoring the displacement of some object using an LVDT over a range 0 to 50 mm. If the full analogue range is encompassed exactly by the available digital codes, then we can calculate the magnitude of the LSB from Equation 3.5:

$$\delta R = \frac{R}{2^n} = \frac{50}{2^{12}} = 0.012 \text{ mm}$$

In this example, the quantization error imposes an accuracy of $\pm\frac{1}{2}\delta R = \pm 0.006$ mm. This presupposes that we use the whole range of available ADC codes. The effective quantization error is clearly worse if only part of the ADC's digitizing range is used. The quantization error indicates the degree of precision that can be attained in an ideal device. It is not, however, representative of the overall accuracy of most real ADCs. We will discuss other sources of inaccuracy later in this chapter.

Quantization noise

For a data-acquisition system equipped with an n-bit ADC and designed to measure signals over a range R, we have seen that the quantization error is $\pm Q$, where $Q = \frac{1}{2}\delta R$. The difference between an analogue value and its digitized representation appears as a varying noise signal superimposed upon the true analogue signal. The amplitude of the noise signal varies by an amount determined by the magnitude of the quantization error and, if the signal to

be digitized consists of a pure sine wave of amplitude $\pm\frac{1}{2}R$, the root-mean-square (rms) value of the noise component is given by:

$$N_{\mathrm{rms}} = \frac{\frac{1}{2}\delta R}{\sqrt{3}} \tag{3.6}$$

which, when we substitute for δR, gives:

$$N_{\mathrm{rms}} = \frac{\frac{1}{2}R}{2^n\sqrt{3}} \tag{3.7}$$

The rms value of the signal itself is:

$$S_{\mathrm{rms}} = \frac{\frac{1}{2}R}{\sqrt{2}} \tag{3.8}$$

so the ratio of the rms signal to rms noise values – the signal-to-noise ratio, SNR – is given by

$$\mathrm{SNR} = \frac{S_{\mathrm{rms}}}{N_{\mathrm{rms}}} = \sqrt{\frac{3}{2}}2^n \tag{3.9}$$

It is normal to express SNR in decibels (dB), where $\mathrm{SNR_{dB}} = 20\log(\mathrm{SNR})$. This gives the approximate relationship:

$$\mathrm{SNR_{dB}} \approx 1.76 + 6.02n \text{ dB} \tag{3.10}$$

This equation relates the number of bits to the dynamic range of the ADC – i.e. the signal-to-noise ratio (SNR) inherent in digitization. Conversely, in a real measuring system, where other sources of noise are present, Equation 3.10 can be used to determine the number of ADC bits that will encode signal changes above the ambient noise level. The contribution made by the low order bits of an ADC may be considerably less than the rms level of noise introduced by other system components. For example, differential and integral non-linearities inherent in the ADC, electronic pickup, sensor noise and unwanted fluctuations in the measurand itself may also degrade the SNR of the system as a whole. In many systems the SNR is limited to around 75 to 85 dB by these factors. Where large noise amplitudes are present, it is fruitless to employ a very high resolution ADC. It may, in such cases, be possible to use an ADC with a lower resolution (and hence lower $\mathrm{SNR_{dB}}$) without losing any useful information. Chapter 4 presents some simple techniques for removing unwanted noise from digitized signals.

Conversion time

Most types of ADC use a multiple-stage conversion process. Each stage might involve incrementing a counter or comparing the analogue signal to some digitally generated approximation. Consequently, analogue-to-digital conversion does not occur instantaneously. Depending upon the method of conversion used, times ranging from a few microseconds up to several seconds may be required. Conversion times are generally quoted in manufacturer's data sheets as the time required to convert a full-scale input. Some devices (such as binary counter type ADCs) are capable of converting lower level signals in a shorter time. In general, low resolution devices tend to be faster than high resolution ADCs. The fastest 16-bit ADCs currently have conversion times of about 1 μs. As a rough rule-of-thumb, the conversion time of the fastest devices currently available tends to increase by roughly an order of magnitude for every additional 2 bits resolution. The conversion times applicable to the various types of ADC are described in the following section.

Types of ADC

There are several basic classes of ADC. The different conversion techniques employed make each type particularly suited to certain types of application. Some ADCs are implemented by using a combination of discrete components (counters, DACs etc.) in conjunction with controlling software. This approach is particularly suited to producing very high resolution converters. However, it tends to be used less often in recent years as high resolution and reasonably priced monolithic ADCs are now becoming increasingly available. The various types of ADC are described below in approximate order of speed: the slowest first.

Voltage-to-frequency conversion ADCs

This type of ADC employs a voltage-to-frequency converter (VFC) to transform the input signal into a sequence of digital pulses. The frequency of the pulse train is proportional to the input voltage. A binary counter counts the pulses over a fixed time interval and the total accumulated count provides the ADC's digital output. The time period over which the pulses are counted varies with the required resolution and full-scale frequency of the VFC. Typical conversion times range from about 50 ms up to several seconds.

Because the input voltage is effectively averaged over the conversion period, VFC-based ADCs exhibit good noise immunity. However, their slow response restricts them to low speed sampling applications. This type of ADC is inherently monotonic, but linearities

and gain errors can be variable. Devices based on lower frequency (10 kHz) VFCs tend to be more accurate than those employing high speed VFCs.

VFCs are sometimes used to digitize analogue signals at remote sensing locations. The advantage of this approach is that sensor signals can be more easily transmitted in digital form over long distances or through noisy environments. The digital pulse train is received by the PC or data-logging unit and then processed using a suitable counter. The resolution and speed of such a system can easily be modified under software control by reprogramming the counter and timer hardware accordingly.

Dual-slope (integrating) ADCs

Dual-slope ADCs each employ a binary counter in conjunction with an integrating circuit that sums the input signal over a fixed time period as shown in Figure 3.11. The rate of increase of the integral during this time is proportional to the average input signal. When the integration has been completed, a negative analogue reference voltage is applied to the integrating circuit and the timer is started. The combined integral of the two inputs then falls linearly. The time taken for the integral to fall to zero is directly proportional to the *average* input voltage. The binary output from the timer is then used to provide the ADC's digital output.

Because the input signal is integrated over time, this type of ADC averages out signal variations due to noise and other sources. Typical

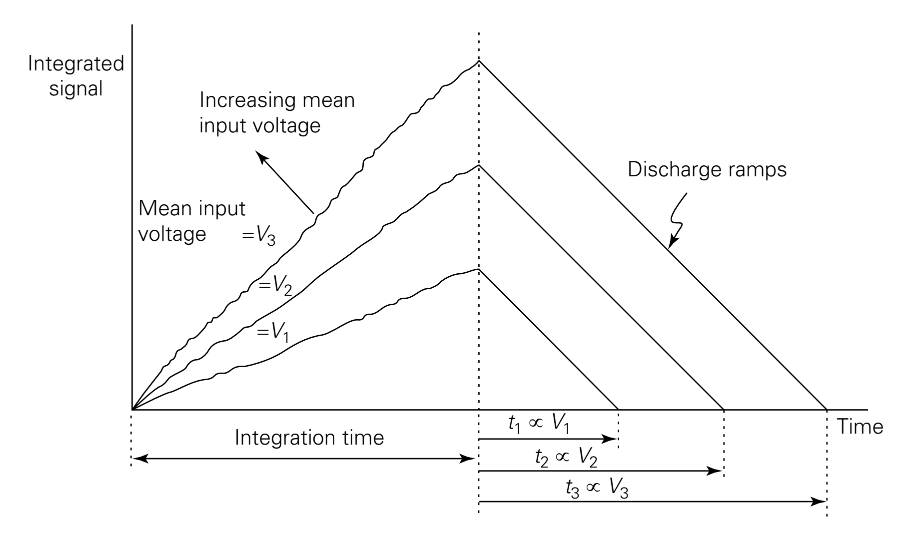


Figure 3.11 *Signal integration in a dual-slope ADC*

integration times are usually of the order of a few milliseconds or longer, limiting the sample rate to typically 5–50 Hz. Dual slope ADCs are particularly suited to use in noisy environments and are often capable of rejecting mains-induced noise. For this reason, they are popular in low speed sampling applications such as temperature measurement. Dual-slope ADCs are relatively inexpensive, offer good accuracy and linearity, and can provide resolutions of typically 12 to 16 bits.

The related single-slope (or Wilkinson) technique involves measuring the time required to discharge a capacitor which initially holds a charge proportional to the input signal. In this case, the capacitor may be a component of circuitry used for signal conditioning or pulse shaping. This technique is sometimes used in conjunction with nuclear radiation detectors for pulse-height analysis in systems designed for X-ray or gamma-ray spectrometry.

Binary counter ADCs

This type of ADC also employs a binary counter, but in this case it is connected to the input of a DAC. The counter is supplied with a clock input of fixed frequency. As the counter is incremented it causes the analogue output from the DAC to increase as shown in Figure 3.12(a). This output is compared with the signal to be digitized and, as soon as the DAC's output reaches the level of the input signal, the counter is stopped. The contents of the counter then provide the ADC's digital output. The accuracy of this type of converter depends upon the precision of the DAC and the constancy of the clock input. The binary counting technique provides moderately good resolution and accuracy, although conversion times

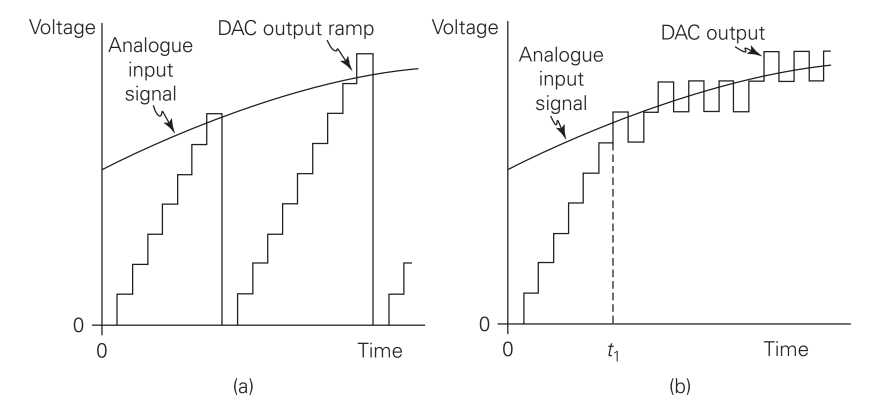


Figure 3.12 *DAC output generated by (a) binary counter ADCs and (b) tracking ADCs*

can be quite long, particularly for inputs close to the upper end of the device's measuring range. This limits throughput to less than a few hundred samples per second.

The main disadvantage with this type of converter is that the conversion time varies with the magnitude of the input signal. A variant of the simple binary counter method, known as the tracking converter, provides a solution to this problem and also allows higher sampling rates to be used. The tracking converter continuously follows the analogue input, ramping its DAC output up or down to maintain a match between its digital output and the analogue input as shown in Figure 3.12(b). The software may, at any time after t_1, stop the tracking (which temporarily freezes the digital output) and then read the ADC. After an initial conversion has been performed, subsequent conversions only require enough time to count up or down to match any (small) change in the input signal. This method operates at a somewhat faster (and less variable) speed than the simple binary counter ADC.

Successive approximation ADCs

The successive approximation technique makes use of a DAC to convert a series of digital approximations of the input signal into analogue voltages. These are then compared with the input signal. The approximations are applied in a binary-weighted sequence as shown in Figure 3.13 which, for the sake of clarity, shows only a 4-bit successive approximation sequence. Eight to 16 bits are more typical of actual ADC implementations.

A reference voltage corresponding to the ADC's MSB is generated first. If this is less than the input signal, a 1 bit is stored in the MSB position of an internal Successive Approximation Register (SAR), otherwise a 0 is stored. Each subsequent approximation involves generating a voltage equivalent to all of the bits in the SAR which have so far been set to 1, plus the value of the next bit in the sequence. Again, if the total voltage is less than the input signal, a 1 value is stored in the appropriate bit position of the SAR. The process repeats, for bits of lesser significance until the LSB has been compared. The SAR will then contain a binary approximation of the analogue input signal.

Because this process involves only a small number of iterations (equal to the number of bits), successive approximation ADCs can operate relatively quickly. Typical conversion times are of the order of 5–30 μs. Successive approximation ADCs offer between 8- and 16-bit resolutions and exhibit a moderately high degree of linearity. This type of ADC is widely used in PC interfacing applications for data acquisition at rates up to 100 kHz. Many manufacturers

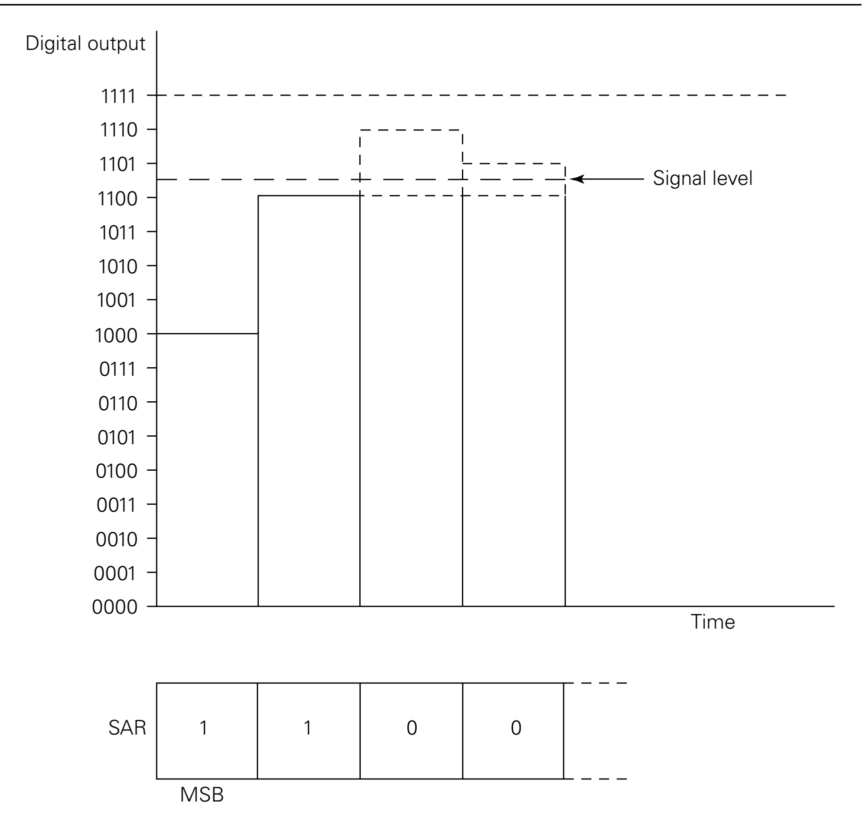


Figure 3.13 *DAC output generated during successive approximation*

provide inexpensive general-purpose DA&C cards based on successive approximation ADCs.

Unlike some other types of ADC, the process of successive approximation does not involve an inherent averaging of the input signal. The main characteristic of these devices is their high operating speed rather than noise immunity. To fully utilize this high speed sampling capability, the ADC's input must remain constant during the conversion. Many ADC cards employ on-board S/H circuits to freeze the input until the conversion has been completed. Some monolithic successive approximation ADCs include built-in S/H circuits for this purpose. In these cases the total conversion time specified in manufacturer's data sheets *may* include the acquisition time of the S/H circuit.

Parallel (flash) ADCs

This is the fastest type of ADC and is normally used in only very high speed applications, such as in video systems. It employs a network

of resistors which generate a binary-weighted array of reference voltages. One reference voltage is required for each bit in the ADC's digital output. A comparator is also assigned to each bit. Each reference voltage is applied to the appropriate comparator, along with a sample of the analogue input signal. If the signal is higher than the comparator's reference voltage, a logical 1 bit is generated, otherwise the comparator outputs a logic 0.

In this way the signal level is simultaneously compared with each of the reference voltages. This parallel digitization technique allows conversions to be performed at extremely high speed. Conversion times may be as low as a few ns, but more typically fall within the range 50–1000 ns. Parallel converters require multiple comparators and this means that high resolution devices are difficult and expensive to fabricate. Resolutions are consequently limited to 8 to 10 bits or less. Greater resolutions can sometimes be achieved by cascading two flash converters. Some pseudo-parallel converters, known as subranging converters, employ a half flash technique in which the signal is digitized in two stages (typically within about 1 μs). The first stage digitizes the most significant bits in parallel. The second stage digitizes the least significant bits.

Using ADCs

As well as their analogue input and digital output lines, most monolithic ADCs have two additional digital connections. One of these, the Start Conversion (also sometimes known as the SC or START) pin, initiates the analogue-to-digital conversion process. Upon receiving the SC signal, the ADC responds by *deactivating* its End of Conversion (EOC) pin and then, when the conversion process has been completed, it asserts EOC once more. The processor should sense the EOC signal and then read the digitized data from the ADC's output register.

On plug-in DA&C cards, the SC and EOC pins are generally mapped to separate bits within one of the PC's I/O ports and can thus be controlled and sensed using assembly language `IN` and `OUT` instructions. The ADC's output register is also normally mapped into the PC's I/O space. In contrast, stand-alone data-logging units and other intelligent instruments may initiate and control analogue-to-digital conversion according to preprogrammed sequences. In these cases ADC control is reduced to simply issuing the appropriate high level commands from the PC.

As an alternative to software initiation, some systems allow the SC pin to be controlled by on-board components such as counters, timers or logic level control lines. Some ADC cards include a provision for the EOC signal to drive one of the PC's interrupt request

lines. Such systems allow the PC's software to start the conversion process and then to continue with other tasks rather than waiting for the ADC to digitize its input. When the conversion is complete the ADC asserts EOC, invoking a software interrupt routine which then reads the digitized data.

Most ADC cards will incorporate I/O-mapped registers which control not just the ADC's SC line, but will also operate an on-board multiplexer and S/H circuit (if present) as shown in Figure 3.14. The details of the register mapping and control-line usage vary between different systems, but most employ facilities similar to those described above. Often the S/H circuit on the input to the ADC is operated automatically when the SC line is asserted. It should be noted, however, that simultaneous S/H circuits are generally operated independently of the ADC via separate control lines. You should consult your system's technical documentation for precise operational details.

ADC characteristics and errors

Figure 3.15 illustrates the characteristics of an ideal ADC. For the sake of clarity, the output from a hypothetical 3-bit ADC is shown. The voltage supplied to the ADC's input is expressed as a fraction of the full-scale input, FS.

Note that each digital code can represent a range of analogue values known as the code width. The analogue value represented by each binary code falls at the mid-point of the range of values encompassed by that code. These mid-range points lie on a straight line passing through the origin of the graph as indicated in the figure. Consequently the origin lies at the mid-range point of the lowest quantum. In this illustration, a change in input equivalent to only $\frac{1}{2}$ LSB will cause the ADC's output to change from 000b to 001b. Because of the positions of the zero and full-scale points, only $2^n - 1$ (rather than 2^n) *changes* in output code occur for a full-scale input swing.

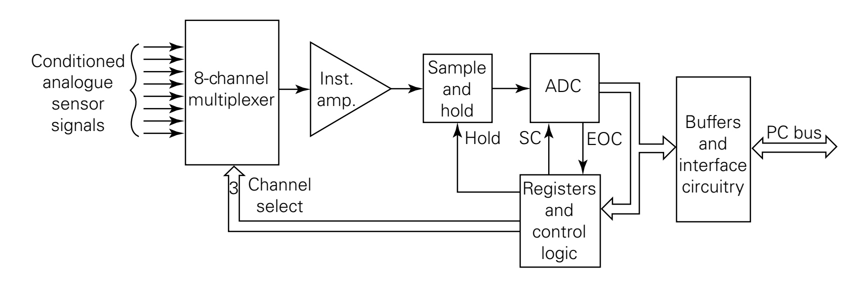


Figure 3.14 *A typical multiplexed ADC card*

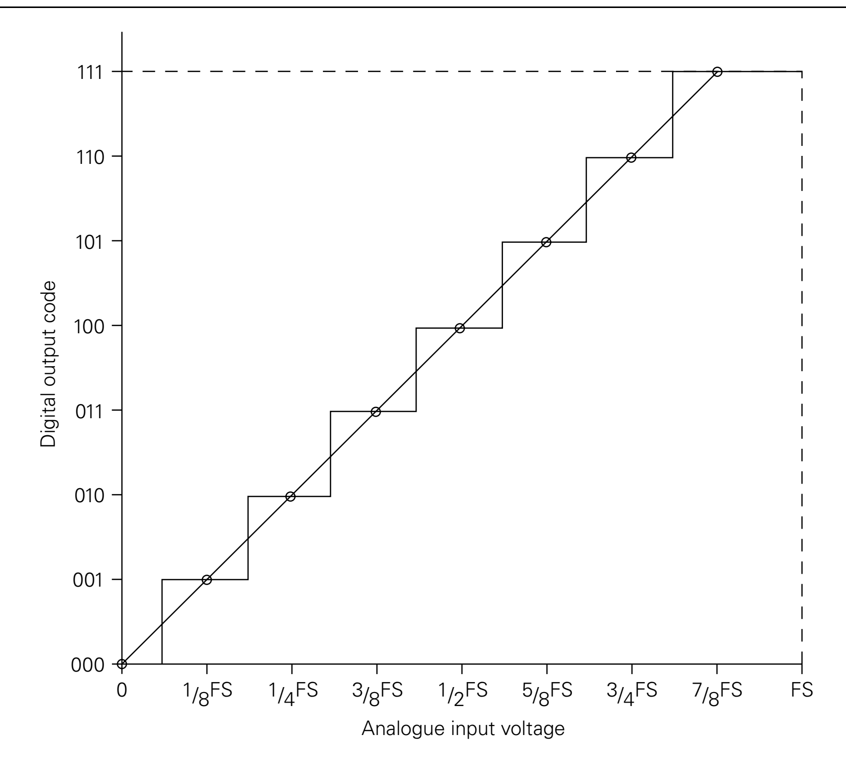


Figure 3.15 *Transfer characteristic of an ideal ADC*

Like DACs, analogue-to-digital converters exhibit several forms of non-ideal behaviour. This often manifests itself as a gain error, offset error or non-linearity. Offset and gain errors present in ADCs are analogous to the corresponding errors already described for DACs. These are illustrated in Figure 3.16 which, for the sake of clarity, shows only the centre points of each code. ADC gain errors can be caused by instabilities in the ADC's analogue reference voltage or by gain errors in their constituent DACs. Gain and offset errors in most monolithic ADCs are very small and can often be ignored.

ADCs may have missing codes – i.e. they may be incapable of generating some codes between the end points of their measuring range. This occurs if the DAC used within the ADC is non-monotonic. Non-linearity (sometimes referred to as integral non-linearity) is a measure of the maximum deviation of the actual transfer characteristic from the ideal response curve. Non-linearities are usually quoted as a fraction of the LSB. If an ADC has a non-linearity of less than $\frac{1}{2}$ LSB then there is no possibility that it will have missing codes.

Differential non-linearity is the maximum difference between the input values required to produce any two consecutive changes in the

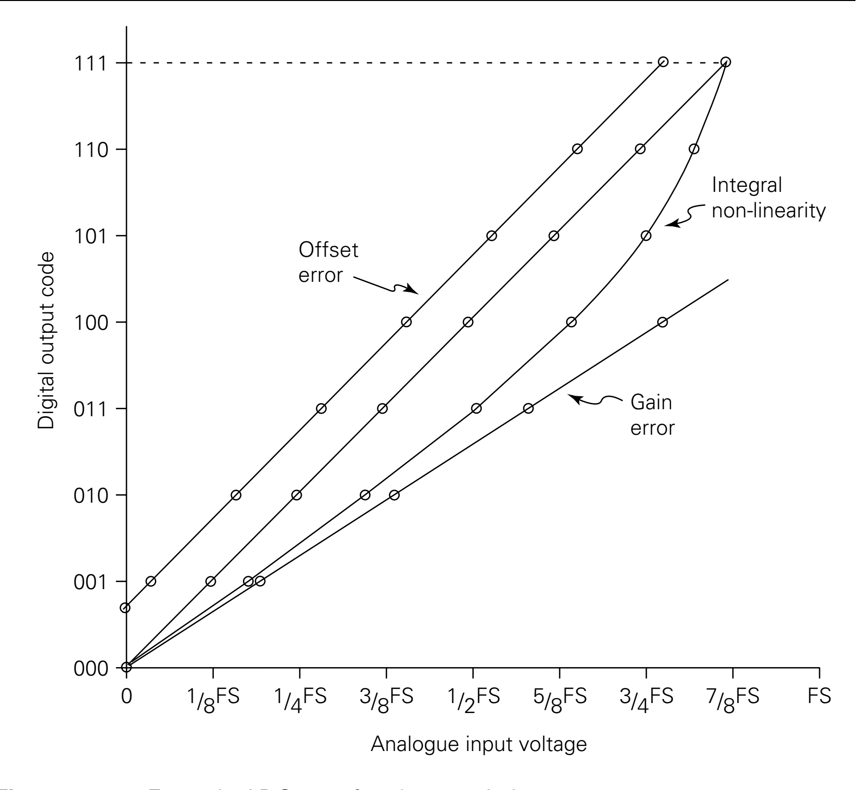


Figure 3.16 *Errors in ADC transfer characteristics*

digital output – i.e. the maximum deviation of the code width from its ideal value of 1 LSB. Non-linearities often occur when several bits all change together (e.g. as in the transition from 255 to 256) and because of this they tend to follow a repeated pattern throughout the converter's range.

The overall accuracy of an ADC will be determined by the sum total of the deviations from the ideal characteristic introduced by gain errors, offset errors, non-linearities and missing codes. These errors are generally temperature dependent. Gain and offset errors can sometimes be trimmed or removed, but non-linearities and missing codes cannot be easily compensated for. Accuracy figures are often quoted in ADC data sheets. They are usually expressed as a percentage of full-scale input range or in terms of the analogue equivalent of the LSB step size. Typical accuracy figures for 12-bit monolithic ADCs are generally of the order of $\pm\frac{1}{2}$ to ±1 LSB. However, these figures may be significantly worse (perhaps 4 to 8 LSB in some cases) at the extremes of the ADC's working temperature range. You are advised to study carefully manufacturers' literature

in order to determine the operational characteristics of the ADC in your own system.

3.6 Analogue measurements

In this section we will discuss three topics of particular importance in the design of analogue measuring systems: accuracy, amplification and throughput.

Accuracy

The accuracy of the whole measuring system will be determined, not just by the precision of the ADC, but also by the accuracy and linearity of the sensor and signal-conditioning circuits used. Random or periodic noise will also affect the measurement accuracy, introducing either statistically random or systematic uncertainties. The inaccuracies inherent in each component of the system (e.g. sensor instabilities, amplifier gain errors, S/H accuracy, ADC quantization error and linearity) should be carefully assessed and summed with the expected (or measured) noise levels in order to arrive at the total potential error. A simple arithmetic sum will provide an estimate of the maximum possible error. However, in some measurements, the errors might be combined such that they oppose each other and tend to cancel out. A figure more representative of the average error which is likely to occur – i.e. the statistical root-sum-square (rss) error – can be obtained by adding the individual errors in quadrature, as follows:

$$\varepsilon = \sum_{k=0}^{k=j} \delta_k^2 \qquad (3.11)$$

Here, ε is the rss error (equivalent to the standard deviation of many readings of a fixed input), j is the number of sources of error and δ_k is the kth source of error expressed either in terms of the units of the measurand or as a fraction of the full-scale measurement range. To simplify the calculation δ_k contributions of less than about one-quarter of the maximum δ_k can usually be ignored without significantly affecting the result. Typical errors introduced by S/H, multiplexer and amplifiers (assuming that they are allowed to settle adequately) are often of the order of ± 0.01 per cent of full scale, or less. This may be a significant source of error, particularly in high resolution systems (i.e. those using ADCs of greater than 10 bits resolution).

Amplification and extending dynamic range

The conversion accuracy of an ADC is ultimately limited by the device's resolution. Unless the range of signal levels generated by the signal-conditioning circuitry is accurately matched to the ADC's full-scale range (typically up to 5 or 10 V), a proportion of the available conversion codes will be unused. In order to take full advantage of the available resolution it is necessary to scale the signal by means of suitable amplifying components. This can easily be accommodated using fixed gain operational amplifiers or instrumentation amplifiers. Many proprietary PC data-acquisition cards incorporate amplifiers of this kind. The gain can generally be selected by means of jumpers or DIP switches when the device is installed in the PC. This approach is ideal if the system is intended to measure some signal over a fixed range to a predetermined degree of accuracy.

However, many sensors have wide dynamic ranges. LVDT displacement sensors, for example, have a theoretically infinite resolution. With suitable signal conditioning they can be used to measure displacements either over their full-scale range or just over a very small proportion of their range. To measure displacements to the same *fractional* accuracy over full or partial ranges, it is necessary to dynamically vary the gain of the signal-conditioning circuit. This is generally accomplished by means of Programmable-Gain Amplifiers (PGAs).

The gain of a PGA can be selected, from a set of fixed values, under software control. In the case of plug-in ADC cards, gain selection is usually effected by writing a suitable bit pattern via an I/O port to one of the card's control registers. It is possible to maximize the dynamic range of the system by selecting an appropriate PGA gain setting.

The software must, of course, compensate for changes in gain by scaling the digitized readings appropriately. Binary gain ranges (e.g. 1×, 2×, 4×, 8× etc.) are the simplest to accommodate in the software since, to reflect the gain range used, the digitized values obtained with the lowest gains can be simply shifted left (i.e. multiplied) within the processor's registers by an appropriate number of bits. If systems with other gain ranges are used it becomes necessary to employ floating-point arithmetic to adjust the scaling factors.

Amplifiers may produce a non-zero voltage (known as an offset voltage) when a zero-volt input is applied. This can be cancelled by using appropriate trimming components. However, these components can be the source of additional errors and instabilities (such as temperature-dependent drifts) and, because of this, a higher

degree of stability can sometimes be obtained by cancelling the offset purely in software. Offsets can also arise from a variety of other sources within the sensor and signal-conditioning circuits. It can be very convenient to compensate for all of these sources in one operation by configuring the software to measure the total offset and to subtract it from each subsequent reading. If you adopt this approach, you should bear in mind that the input to a PGA from previous amplification stages or signal-conditioning components still possesses a non-zero offset. Changing the gain of the PGA can also affect the magnitude of offset presented to the ADC. It is, therefore, prudent for the software to rezero such systems whenever the PGA's gain is changed.

One of the most useful capabilities offered by PGAs is autoranging. This permits the optimum gain range to be selected even if the present signal level is unknown. An initial measurement of the signal is obtained using the lowest (e.g. 1×) gain range. The gain required to give the optimum resolution is then calculated by dividing the ADC's resolution (e.g. 4096 in the case of a 12-bit converter) by the initial reading. The gain range less than or equal to the optimum gain is then selected for the final reading. This technique obviously reduces throughput as it involves twice as many analogue-to-digital conversions and repeated gain changes.

Throughput

The throughput of an analogue measuring system is the rate at which the software can sample analogue input channels and process the acquired data. It is more conveniently expressed as the number of channels read per second. The distinction between this and the rate at which multiplexed *groups* of sensor channels can be scanned should be obvious to the reader. A system scanning a group of eight sensor channels 50 times per second will have a throughput figure of 400 channels per second.

A number of factors affect throughput. One of the most important of these is the ADC's conversion time, although it is by no means the only consideration. The acquisition time of the S/H circuit, the settling times of the multiplexer, S/H, PGA and other components, the bandwidth of filters, and the time constant of the sensor may all have to be taken into account. Each component must be fast enough to support the required throughput.

When scanning multiple channels, throughput can sometimes be maximized by changing the multiplexer channel as soon as the S/H circuit is switched into hold mode. This allows analogue-to-digital conversion to proceed while the multiplexer's output settles to the

level of the next input channel. This technique, known as overlap multiplexing, requires well-designed DA&C hardware to avoid feed-through between the two channels. Compare this with the usual (slower) technique of serial multiplexing, where each channel is selected, sampled and digitized in sequence.

Throughput is, of course, also limited by the software used. Unless special software and hardware techniques, such as Direct Memory Access (DMA), are employed, each read operation will involve the processor executing a sequence of IN and OUT instructions. These are needed in order to operate the multiplexer (and possibly S/H), to initiate the conversion, check for the EOC signal, read one or two bytes of data and then to store that data in memory. The time required will vary between different types of PC, but on a moderately powered system, these operations will generally introduce delays of several tens of microseconds per channel. Provided that no other software processing is required, a fast (e.g. successive approximation) ADC is used, and that the bandwidth of the signal-conditioning circuitry does not limit throughput, a well-designed 80486-based data-acquisition system might be capable of reading several thousand channels per second. Systems optimized for high speed sampling of single channels can achieve throughput rates in excess of 10 000–20 000 samples per second.

Most systems, however, require a degree of additional real-time processing. The overheads involved in scaling or linearizing the acquired data or in executing control algorithms will generally reduce the maximum attainable throughput by an order of magnitude or more. Certain operations, such as updating graphical displays or writing data to disk can take a long (and possibly indeterminate) time. The time needed to update a screen display, for example, ranges from a few milliseconds up to several hundred milliseconds (or even several seconds), depending upon the complexity of the output. Speed can sometimes be improved by coding the time-critical routines in assembly language rather than in C, Pascal or other high level languages.

In assessing the speed limitations which are likely to be imposed by software, it is wise to perform thorough timing tests on each routine that you intend to use during the data-acquisition period. In many cases, raw data can be temporarily buffered in memory for subsequent processing during a less time-critical portion of the program. By carrying out a detailed assessment of the timing penalties associated with each software operation you should be able to achieve an optimum distribution of functionality between the real-time and post-acquisition portions of the program.

3.7 Timers and pacing

Most real-time applications require sensor readings to be taken at precise times in the data-acquisition cycle. In some cases, the time at which an event occurs, or the time between successive events, can be of greater importance than the attributes of the event itself. The ability to pace a data-acquisition sequence is clearly important for accurately maintaining sampling rates and for correct operation of digital filters, PID algorithms and time-dependent (e.g. chart recorder) displays. A precise timebase is also necessary for measurement of frequency, for differentiating and integrating sensor inputs, and for driving stepper motors and other external equipment.

Timing tasks can be carried out by using counters on an adaptor card inserted into one of the PC's expansion slots. Indeed many analogue I/O cards have dedicated timing and counting circuitry, which can be used to trigger samples, to interrupt the PC, to control the acquisition of a preprogrammed number of readings or to generate waveforms.

Another approach to measuring elapsed time is to use the timing facilities provided by the PC. This is a relatively easy task when programming in a real-mode environment (e.g. DOS). It becomes more complex, however, under multitasking operating systems such as Windows NT or OS/2, where one has limited access to, and less control over, the PC's timing hardware. The PC is equipped with a programmable system clock based on the Intel 8254 timer counter, as well as a Motorola MC146818A Real Time Clock (RTC) IC. These, together with a number of BIOS services provide real-mode programs with a wealth of timing and calendrical features.

Whatever timing technique is adopted, it is important to consider the granularity of the timing hardware – i.e. the smallest increment in time that it can measure. This should be apparent from the specification of the timing device used. The PC's system timer normally has a granularity of about 55 ms and so (unless it is reprogrammed accordingly) it is not suitable for measuring very short time intervals. The RTC provides a periodic timing signal with a finer granularity: approximately 976 μs. There are various software techniques that can yield granularities down to less than 1 μs using the PC's hardware, although such precise timing is limited in practice by variations in execution time of the code used to read the timer. The texts by van Gilluwe (1994) and Sanchez and Canton (1994) provide useful information for those readers wishing to exploit the timing capabilities of the PC.

When devising and using any timing system that interacts with data-acquisition software (as opposed to a hardware-only system), it must be borne in mind that the accuracy of time measurements will be determined, to a great extent, by how the timing code is implemented. As in many other situations, assembly language provides greater potential for precision than a high level language. A compiled language such as C or Pascal is often adequate for situations where timing accuracies of the order of 1 ms are required.

Most programming languages and development environments include a variety of time-related library functions. For example, National Instruments' LabWindows/CVI (an environment and library designed for creating data-acquisition programs) when running on Windows NT supplies the application program with a timing signal every 1 ms or 10 ms (depending upon configuration). A range of elapsed-time, time-delay, and time-of-day functions is also provided.

Watchdog timers

In many data-acquisition applications the PC must communicate with some external entity such as an intelligent data-logging module or a programmable logic controller. In these cases it can be useful for both components of the system to be 'aware' of whether the other is functioning correctly. There are a number of ways in which the state of one subsystem can be determined by another. A program running on a PC can close a normally open contact to indicate that it has booted successfully and is currently monitoring some process or other. If the PC and relay subsequently lose power, the contact will open and alert external equipment or the operator to the situation. However, suppose that power to the PC remained uninterrupted, but the software failed due to a coding error or memory corruption. The contact would remain closed even though the PC was no longer functional. The system could not then make any attempt to automatically recover from the situation. Problems like this are potentially expensive, especially in long-term data-logging applications where the computer may be left unattended and any system crash could result in the loss of many days' worth of data.

A watchdog timer can help to overcome these problems. This is a simple analogue or digital device which is used to monitor the state of one of the component parts of a data-acquisition or computer system. The subsystem being monitored is required to refresh the watchdog timer periodically. This is usually done by regularly pulsing or changing the state of a digital input to the watchdog timer. In some implementations the watchdog generates a periodic timing

signal and the subsystem being monitored must then refresh the watchdog within a predetermined interval after receipt of this signal. If the watchdog is not refreshed within a specified time period it will generate a time-out signal. This signal can be used to reset the subsystem or it can be used for communicating the timeout condition to other subsystems.

The IBM PS/2 range of computers is equipped with a watchdog timer which monitors the computer's system timer interrupt (IRQ0). If the software fails to service the interrupt, the watchdog generates an NMI (see Chapter 5).

It is worth mentioning at this point that you should avoid placing watchdog-refresh routines within a hardware-generated periodic interrupt handler (e.g. the system timer interrupt). In the event of a software failure, it is possible that the interrupt will continue to be generated at the normal rate!

It is sometimes necessary to interface a watchdog timer to a PC-based data-acquisition system in order to detect program crashes or loss of power to the PC. The timeout signal might be fed to a programmable logic controller, for example, to notify it (or the operator) of the error condition. It is also possible to reboot the PC by connecting the timeout signal to the reset switch (present on most PC-compatible machines) via a suitable relay and/or logic circuits. Occasionally, software crashes can (depending upon the operating system) leave the PC's support circuits in such a state of disarray that even a hardware reset cannot reboot the computer. The only solution in this case is to temporarily turn off the computer's power. Although rebooting via the reset switch might be possible, the process can take up to two or three minutes on some PCs. It is not always easy for the software to completely recover from this type of failure, especially if the program crash or loss of power occurred at some critical time such as during a disk-write operation. It is preferable for the software to attempt to return to a default operating mode and not to rely on any settings or other information recorded on disk. The extent to which this is feasible will depend upon the nature and complexity of the application.

4 Sampling, noise and filtering

Virtually all data-acquisition and control systems are required to sample analogue waveforms. The timing of these samples is often critical and has a direct bearing on the system's ability to accurately reconstruct and process analogue signals. This chapter introduces elements of sampling theory and discusses how measurement accuracy is related to signal frequency and to the temporal precision of the sampling hardware. The associated topic of digital filtering is also discussed.

4.1 Sampling and aliasing

Analogue signals from sensors or transducers are continuous functions, possessing definite values at every instant of time. We have already seen that the PC can read only digitized representations of a signal and that the digitization process takes a finite time. Implicit in our discussion has been the fact that the measuring system is able to obtain only discrete samples of the continuous signal. It remains unaware of the variation of the signal between samples.

The importance of sampling rate

We can consider each sample to be a digital representation of the signal at some fixed point in time. In fact, the readings are not truly instantaneous but, if suitable sample-and-hold circuits are used, each reading is normally representative of a very well-defined instant in time (typically accurate to a few nanoseconds).

In general, the sampling process must be undertaken in such a way as to minimize the loss of time-varying information. It is important to take samples at a sufficiently high rate in order to be able to accurately reconstruct and process the signal. It should be obvious that a system which employs too low a sampling rate will be incapable

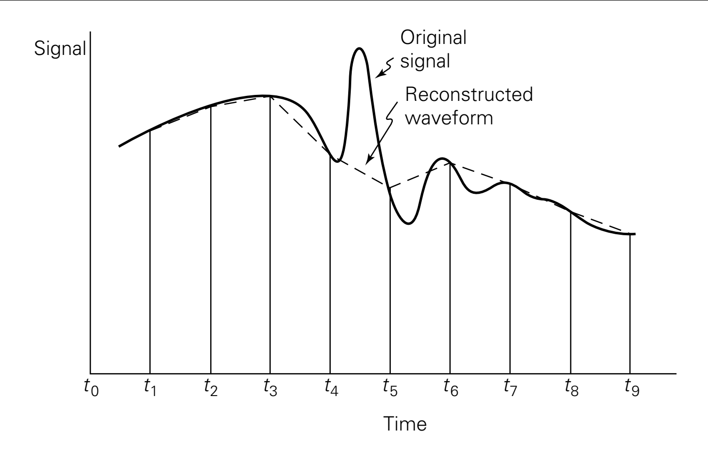


Figure 4.1 *Degradation of a reconstructed signal as the sampling rate is reduced*

of responding to rapid changes in the measurand. Such a situation is illustrated in Figure 4.1. At low sampling rates, the signal is poorly reconstructed. High frequency components such as those predominating between sample times t_4 and t_6 are most badly represented by the sampled points. This can have serious consequences, particularly in systems that have to *control* some process. The inability to respond to transient disturbances in the measurand may compromise the system's ability to maintain the process within required tolerances.

Clearly, the relationship between the sampling rate and the maximum frequency component of the signal is of prime importance. There are normally a number of practical limitations on the maximum sampling frequency that can be achieved: for example, the ADC conversion speed, the execution time of interface software and the time required for processing the acquired data. The total storage space available may also impose a limit on the number of samples that can be obtained within a specified period.

Nyquist's sampling theorem

We need to understand clearly how the accuracy of the sampled data depends upon the sampling frequency, and what effects will result from sampling at too low a rate. To quantify this we will examine the Fourier transforms (i.e. the frequency spectra) of the signal and the sampled waveform.

Typical waveforms from sensors or transducers consist of a range of different frequency components as illustrated in Figure 4.2(a) and (b). If a waveform such as this is sampled at a frequency ν, where $\nu = 1/t$ and t represents the time interval between samples, we obtain the sampled waveform shown in Figure 4.2(c). In the time domain,

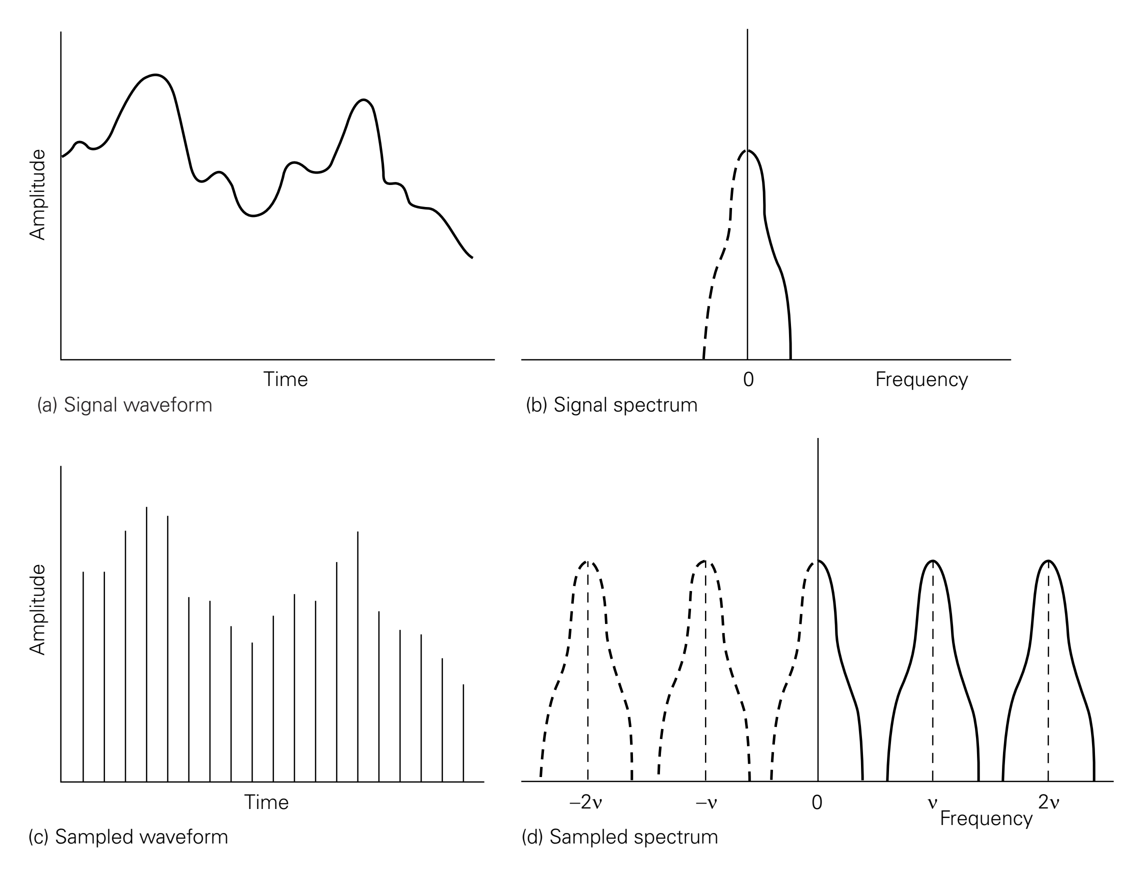


Figure 4.2 *Representation of a sampled waveform in the time and frequency domains*

the sampled waveform consists of a series of impulses (one for each sample) modulated by the actual signal. In the frequency domain (Figure 4.2(d)) the effect of sampling is to cause the spectrum of the signal to be reproduced at a series of frequencies centred at integer multiples of the sampling frequency.

The original frequency spectrum can be easily reconstructed in the example shown in Figure 4.2. It should, however, be clear that as the maximum signal frequency, f_{max}, increases, the individual spectra will widen and begin to overlap. Under these conditions, it becomes impossible to separate the contributions from the individual portions of the spectra, and the original signal cannot then be accurately reproduced. Overlapping occurs when f_{max} reaches half the sampling frequency. Thus, for accurate reproduction of a continuous signal containing frequencies up to f_{max}, the sampling rate, ν, must be greater than or equal to $2f_{max}$. This condition is known as Nyquist's sampling theorem and applies to sampling at a constant frequency. Obviously, sampling using unequal time intervals complicates the detail of the discussion, but the same general principles apply.

Aliasing

Figure 4.2(d) shows that if any component of the signal exceeds $\frac{1}{2}\nu$, the effect of sampling will be to reproduce those signal components at a lower frequency. This phenomenon, known as aliasing, may be visualized by considering an extreme case where a signal of frequency f_{sig} is sampled at a rate equal to f_{sig} (i.e. $\nu = f_{sig}$). Clearly, each sample will be obtained at the same point within each signal cycle and, consequently, the sampled waveform will have a frequency of zero as illustrated in Figure 4.3(a). Consider next the case where f_{sig} is only very slightly greater than ν. Each successive sample will advance by a small amount along the signal cycle as shown in Figure 4.3(b). The resulting train of samples will appear to vary with a new (lower) frequency: one which did not exist in the original waveform! These so-called alias, or beat, frequencies can cause severe problems in systems which perform any type of signal reconstruction or processing – i.e. virtually all DA&C applications.

As a digression, it is interesting to note that some systems (although not usually PC-based DA&C systems) exploit the aliasing phenomenon in order to extract information from high frequency signals. This technique is used in dynamic testing of ADCs and in various types of instrumentation.

In normal sampling applications, however, aliasing is not desirable. It can be avoided by ensuring, first, that the signal is band limited

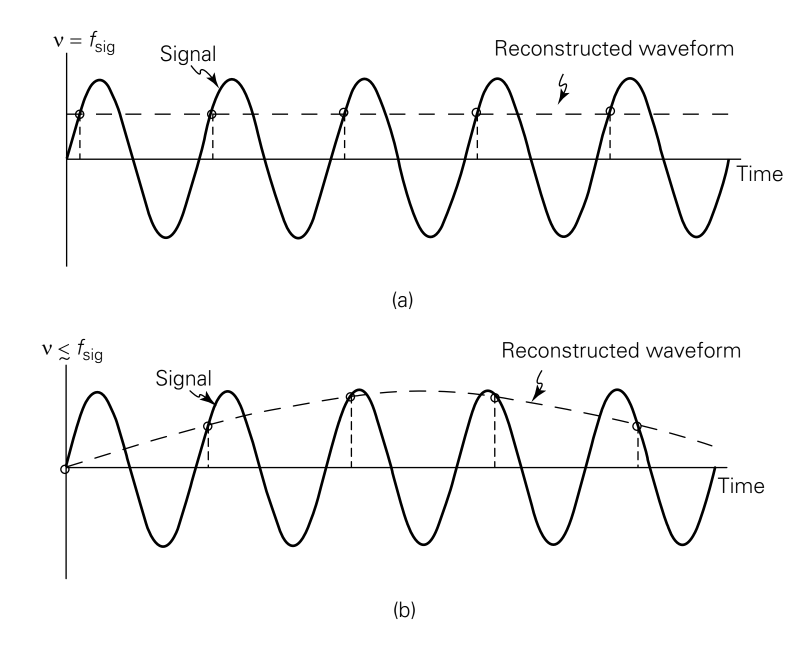


Figure 4.3 *Generation of alias frequencies*

(i.e. has a well-defined maximum frequency, $f_{\max}$) and, second, that the sampling rate, ν, is at least twice $f_{\max}$. It is usual to employ an analogue anti-aliasing low-pass filter in order to truncate the signal spectrum to the desired value of $f_{\max}$ prior to sampling. This results in the loss of some information from the signal, but by judicious selection of the filter characteristics it is usually possible to ensure that this does not have a significant effect on the performance of the system as a whole. Anti-aliasing filters are often an integral part of signal-conditioning units. Strain-gauge-bridge signal conditioners, for example, may incorporate filters with a bandwidth of typically 100 to 200 Hz.

It should be borne in mind that no filter possesses an ideal response (i.e. 100 per cent attenuation above the cut-off frequency, f_0, and 0 per cent attenuation at lower frequencies), although good anti-aliasing filters often possess a steep cut-off rate. Because real filters exhibit a gradual drop in response, it is usually necessary to ensure that ν is somewhat greater than $2f_0$. The sampling rate used will depend upon the form of the signal and upon the degree of precision required. The following figures are provided as a rough guide. Simple one- or two-pole passive anti-aliasing filters may necessitate sampling rates of $5f_0$ to $10f_0$. The steeper cut-off rate attainable with active anti-aliasing filters normally allows sampling at around $3f_0$.

Sampling accuracy

Nyquist's sampling theorem imposes an upper limit on the signal frequencies that can be sampled. However, a number of practical constraints must also be borne in mind. In many applications, the speed of the software (cycling time, interrupt latencies, transfer rate etc.) restricts the sampling rate and hence f_{max}. Some systems perform high speed data capture completely in hardware, thereby circumventing some of the software speed limitations. In these cases, periodic sampling is usually triggered by an external clock signal and the acquired data is channelled directly to a hardware buffer.

The performance of the hardware itself also has a bearing on the maximum frequency that can be sampled with a given degree of accuracy. There is an inherent timing error associated with the sampling and digitization process. This inaccuracy may be a result of the ADC's conversion time or, if a sample-and-hold (S/H) circuit is employed, it may be caused by the circuit's finite aperture time or aperture jitter (see Chapter 3). The amount by which the signal might vary in this time limits the accuracy of the sample and is known as the aperture error.

Consider a time-varying measurand, R. For a given timing uncertainty, δt, the accuracy with which the measurand can be sampled will depend upon the maximum rate of change of the signal. To achieve a given measurement accuracy we must place an upper limit on the signal frequency which the system will be able to sample.

We can express a single frequency (f) component as

$$R = R_0 \sin(2\pi f t) \tag{4.1}$$

The aperture error, A, is defined as

$$A = \frac{dR}{dt}\delta t \tag{4.2}$$

and our sampling requirement is that the aperture error must always be less than some maximum permissible change, δR_{max}, in R, i.e.

$$\frac{dR}{dt} \leq \frac{\delta R_{max}}{\delta t} \tag{4.3}$$

We must decide on a suitable value for δR_{max}. It is usually convenient to employ the criterion: $\delta R_{max} = 1$ LSB (i.e. that A must not exceed 1 LSB). It might be more appropriate in some applications to use different values, however. Applying this criterion, and assuming that the full ADC conversion range exactly encompasses the entire signal

range (i.e. $2R_0$), Equation 4.3 becomes

$$\frac{dR}{dt} \leq \frac{2R_0}{2^n \delta t} \tag{4.4}$$

Here, n represents the ADC resolution (number of bits). Differentiating Equation 4.1, we see that the maximum rate of change R is given by $2\pi f R_0$. Substituting this into Equation 4.4, we obtain the maximum frequency, f_A, that can be sampled with the desired degree of accuracy.

$$f_A = \frac{1}{\pi 2^n \delta t} \tag{4.5}$$

Let us consider a moderately fast, 12-bit ADC with a conversion time of 10 μs. Such a device should be able to accommodate sampling rates approaching 100 kHz. Applying the Nyquist criterion gives a maximum signal frequency of half this (i.e. 50 kHz). However, this criterion only guarantees that, *given sufficiently accurate measuring equipment*, it will be possible to detect this maximum signal frequency. It takes no account of the sampling precision of real ADCs. To assess the effect of finite sampling times we must use Equation 4.5. Substituting the 10 μs conversion time for δt shows that we would be able to sample signal components up to only 7.7 Hz with the desired 1 LSB accuracy! This illustrates the importance of the greater temporal precision achievable with S/H circuits. If we were to employ an S/H circuit, δt could be reduced to the S/H's aperture jitter time. Substituting a typical value of 2 ns for δt shows that, with the benefit of an S/H circuit, the maximum frequency that could be sampled to a 1 LSB accuracy increases to around 39 kHz.

It is often more useful to calculate the actual aperture error resulting from a particular combination of aperture time and signal frequency. Equation 4.2 defines the aperture error. This has its maximum value when R is subject to its maximum rate of change. We have already seen that this occurs when R is zero and that the maximum rate of change of R is $2\pi f R_0$. The maximum possible aperture error, A_{max}, is therefore:

$$A_{max} = 2\pi f \delta t R_0 \tag{4.6}$$

Figure 4.4 depicts values of the ratio $A_{max}/2R_0$ as a function of aperture time and signal frequency.

Reconstruction of sampled signals

The accuracy with which a signal can be sampled is by no means the only consideration. The ability of the DA&C system to precisely

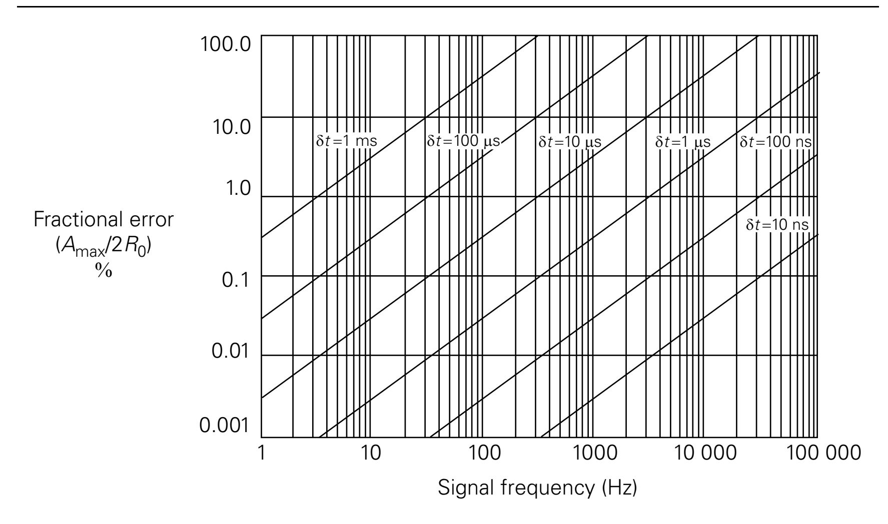


Figure 4.4 *Fractional aperture error as a function of aperture time and signal frequency*

reconstruct the signal (either physically via a DAC or mathematically inside the PC) is often of equal importance. The accuracy with which the sampled signal can be reconstructed depends upon the reconstruction method adopted – i.e. upon the physical or mathematical technique used to interpolate between sampled points.

A linear interpolation (known as first order reconstruction) approximates the signal by a series of straight lines joining each successive sampled data point (see Figure 4.5). This gives a waveform with the correct fundamental frequency together with many additional higher frequency components.

Alternatively, we may interpolate by holding the signal at a fixed value between consecutive points. This is known as zero order reconstruction and is, in effect, the method employed when samples are passed directly to a DAC. In this case, the resulting reconstructed signal will contain a number of harmonics at $\nu \pm f$, $2\nu \pm f$, $3\nu \pm f$ etc. An electronic low-pass filter would be required at the DAC's output in order to remove the harmonics and thereby smoothly interpolate between samples. Note that these harmonics are artefacts of the reconstruction process, not of the sampling process *per se*.

The accuracy of the reconstruction will, of course, depend upon the ratio of the signal and sampling frequencies (ν/f). There is clearly an error associated with each reconstructed point. Ignoring any errors introduced by the sampling mechanism, the reconstruction error will simply be the difference between the reconstructed value and the actual signal value at any chosen instant. In those parts

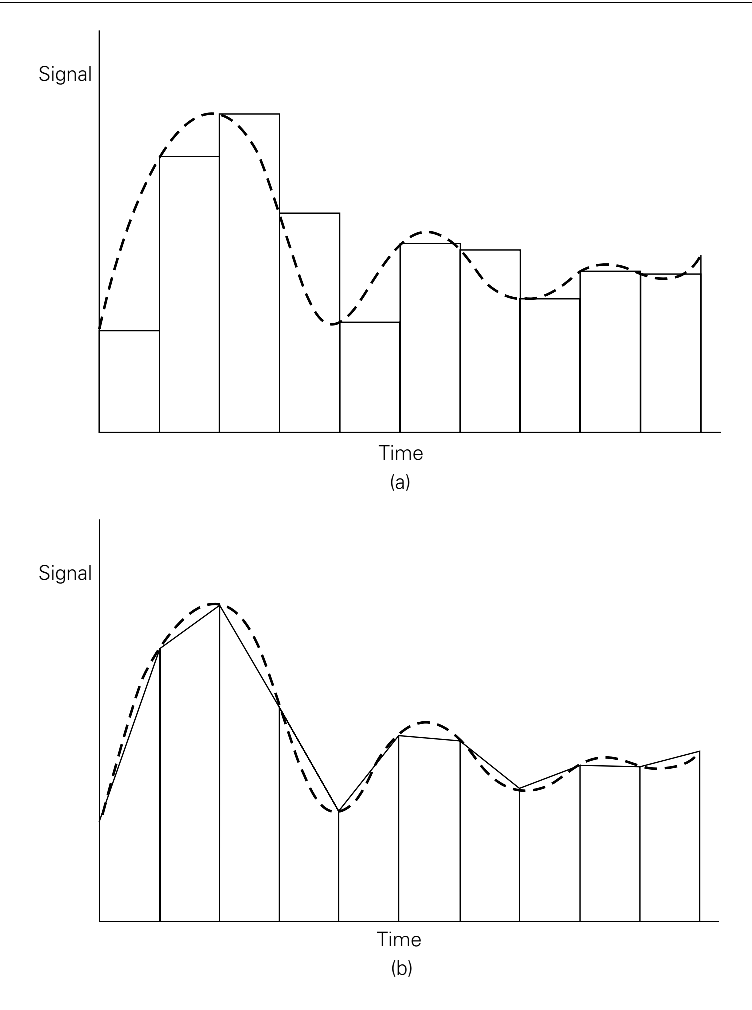


Figure 4.5 *Reconstruction of sampled signals: (a) zero order and (b) first order interpolation*

of Figure 4.5 where high frequency signal components predominate (i.e. where the signal is changing most rapidly), there is a potential for a large difference between the original and reconstructed waveforms. The reconstructed waveform will model the original sampled waveform more accurately if there are many samples per signal cycle.

The values of the average and maximum errors associated with the reconstruction are generally of interest to DA&C system designers. It is a trivial matter to derive an analytical equation for the maximum error associated with a zero order reconstruction, but the calculations necessary to determine the *average* errors can be somewhat more

Table 4.1 *Coefficients of Equation 4.7*

Order	*Desired calculation*	*p*	*q*
Zero	Maximum error	3.1	−1
Zero	Average error	2.0	−1
First	Maximum error	4.7	−2
First	Average error	2.0	−2

involved. For this reason we will simply quote an empirical relation. The following formula can be used to estimate the magnitudes of the maximum and the average fractional errors (E_r) involved in both zero and first order reconstruction.

$$E_r \approx p \left(\frac{\nu}{f} \right)^q \times 100\% \tag{4.7}$$

The coefficients of the equation, p and q, depend upon the order of reconstruction and whether the average or maximum reconstruction error is being calculated. These coefficients are listed in Table 4.1. Do bear in mind that Equation 4.7 is not a precise analytical formula. It should only be used as a rough guide for values of ν/f greater than about 10.

Note that the sampling rate required to achieve a desired degree of accuracy with zero order reconstruction may be several orders of magnitude greater than that necessary with first order interpolation. For this reason, first order techniques are to be preferred in general. Appropriate filtering should also be applied to DAC outputs to minimize zero order reconstruction errors.

In summary, the accuracy of the sampled waveform and the presence of any sampling artefacts will depend upon how the sampled data is processed. Also, the extent to which any such artefacts are acceptable will vary between different applications. All of these points will have a direct bearing on the sampling rate used and must be considered when designing a DA&C system.

Selecting the optimum sampling rate

In designing a DA&C system, we must assess the effect of ADC resolution, conversion time and S/H aperture jitter, as well as the selected sampling rate on the system's ability to achieve some desired level of precision. For the purposes of the present discussion, we will ignore any inaccuracies in the sensor and signal-conditioning circuits, but we must bear in mind that, in reality, they may affect

the accuracy of the system as a whole. We will concentrate here upon sampling rate and its relationship to frequency content and filtering of the signal. In this context, the following list outlines the steps required to ensure that a DA&C system meets specified sampling-precision criteria.

1. First, assess the static precision of the ADC (i.e. its linearity, resolution etc.) using Equations 3.5 and 3.10 to ensure that it is capable of providing the required degree of precision when digitizing an unchanging signal.
2. Assess the effect of sampling rate on the accuracy of signal reconstruction using Equation 4.7. By this means, determine the minimum practicable sampling rate, ν, needed to reproduce the highest frequency component in the signal with the required degree of accuracy. Also bear in mind Nyquist's sampling theorem and the need to avoid aliasing. From ν, you should be able to define upper limits for the ADC conversion time and software cycle times (interrupt rates or loop-repeat rates etc.). Ensure that the combination of software routines and DA&C/computer hardware are actually capable of achieving this sampling rate. Also ensure that appropriate anti-aliasing filters are employed to remove potentially troublesome high frequencies.
3. Given the sample rate, the degree of sampling accuracy required and the ADC resolution, n, use Equations 4.3 to 4.5 to define an upper limit on δt and thereby ensure that the digitization and S/H components are capable of providing the necessary degree of sampling precision.

4.2 Noise and filtering

Noise can be problematic in analogue measuring systems. It may be defined as any unwanted signal component that tends to obscure the information of interest. There are a variety of possible noise sources, such as electronic noise or electromagnetic interference from mains or high frequency digital circuits. These sources tend to be most troublesome with low level signals such as those generated by strain gauges and thermocouples. Additionally, noise may also arise from *real* variations in some physical variable – e.g. unwanted vibrations in a displacement measuring system or temperature fluctuations due to convection and turbulence in a furnace. As we have seen in Chapter 3, the approximations involved in the digitization process are also a source of noise. The presence of noise can be very problematic in some applications. It can make displays appear unsteady,

obscure underlying signal trends, erroneously trigger comparators and seriously disrupt control systems.

It is always good practice to attempt to exclude noise at its source rather than having to remove it at a later stage. Steps can often be taken, particularly with cables and shielding, to minimize noise amplitudes. This topic is discussed briefly in Chapter 3 and further guidance may be found in the text by Tompkins and Webster (1988) or in various manufacturers' application notes and data books, such as Burr Brown's *PCI Handbook* (1988). However, even in the best designed systems, a certain degree of noise pickup is often inevitable. If residual noise amplitudes are likely to have a significant effect on the accuracy of the system, the signal-to-noise ratio must be improved before the underlying signal can be adequately processed. This can be accomplished by using simple passive or active analogue filter circuits. Filtering can also be performed digitally by using suitable software routines.

Software techniques have a number of advantages over hardware filters. Foremost amongst these is flexibility. It is very simple to adjust the characteristics of a digital filter by modifying one or two parameters of the filtering algorithm. Another benefit is that digital filters are more stable and do not exhibit any dependence on environmental factors such as temperature. They are also particularly suited to use at very low frequencies, where hardware filters may be impracticable due to their size, weight or cost. In addition, they are the only way of removing noise introduced by the ADC circuitry during digitization.

Filtering of acquired data can be performed after the data-acquisition cycle has been completed. In some ways this approach is the simplest, as the complete data set is available and the filtering algorithm can be easily adjusted to optimize noise suppression. There are many techniques for post-acquisition filtering and smoothing of data. Most are based on Fourier methods and are somewhat mathematical. They are classed as data-analysis techniques and, as such, fall beyond the scope of this book. Press *et al.* (1992) describe a number of post-acquisition filtering and smoothing techniques in some detail.

Post-acquisition filtering is of little use if we need to base real-time decisions or control signals on a filtered, noise-free signal. In this case we must employ real-time filtering algorithms, which are the topic of this section. The design of real-time digital filters can also be quite involved and requires some moderately complex mathematics. However, this section refrains from discussing the mathematical basis of digital filters and, instead, concentrates on the practical implementation of some simple filtering algorithms. While

the techniques presented will not be suitable for every eventuality, they will probably cover a majority of DA&C applications. Digital filters can generally be tuned or optimized at the development stage or even by the end user and, for this purpose, a number of empirical guidelines are presented to aid in filter design.

Designing simple digital filters

It is impossible for DA&C software to determine the relative magnitudes of the signal and noise encapsulated in a *single* isolated reading. Within *one* instantaneous sample of the total signal-plus-noise voltage, the contribution due to noise is indistinguishable from that due to the signal. Fortunately, when we have a series of samples, noise and signal can often be distinguished on the basis of their frequencies. They usually have different frequency characteristics, each existing predominantly within well-defined frequency bands. By comparing and combining a series of readings it is possible to ascertain what frequencies are present and then to suppress those frequencies at which there is only noise (i.e. no signal component). The process of removing unwanted frequencies is known as filtering.

Signal and noise characteristics

Many signals vary only slowly. We have already seen in Chapter 3 that some types of sensor and signal-conditioning circuits have appreciable time constants. Noise, on the other hand, may occur at predominantly one frequency (e.g. the mains 50/60 Hz frequency) or, more often, in a broad band as shown in Figure 4.6. The signal frequencies obtained with most types of sensor will generally be

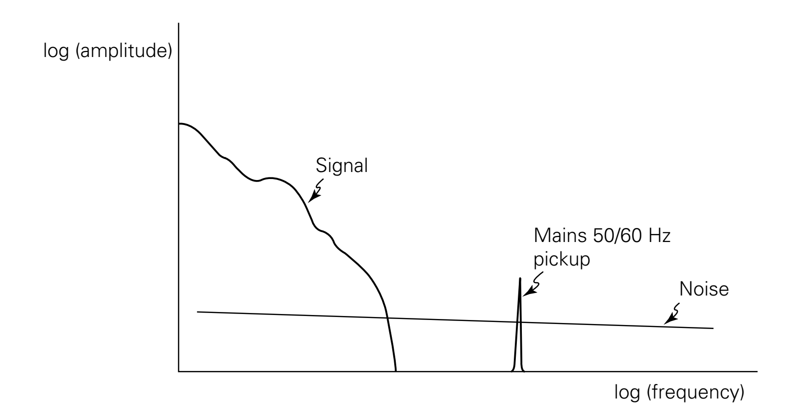


Figure 4.6 *Typical noise and signal spectra*

quite low. On the other hand, noise due to radiated electromagnetic pickup or from electronic sources often has a broad spectrum extending to very high frequencies. This high frequency noise can be attenuated by using an appropriate low-pass filter (i.e. one which suppresses high frequencies while letting low frequencies pass through unaffected). Noise might also exist at low frequencies, overlapping the signal spectrum. Because it occupies the same frequencies as the signal itself, this portion of the noise spectrum cannot be filtered out without also attenuating the signal.

When designing a digital filter, it is advisable to first determine the principal sources of noise in the system and to carefully assess the noise and signal spectra present. Such an exercise provides an essential starting point for determining which frequency bands you wish to suppress and which bands you will need to retain.

Filter characteristics

Low-pass filters attenuate all frequencies above a certain cut-off frequency, f_0, while leaving lower frequencies (virtually) unaffected. Ideally, such filters would have a frequency characteristic similar to curve (a) shown in Figure 4.7. In practice, this is impossible to achieve, and filter characteristics such as that indicated by curve (b) are more usually obtained with either electronic or digital (software) filters. Other filter characteristics are sometimes useful. High-pass filters (curve (c)), for example, suppress frequencies lower than some cut-off frequency while permitting higher frequencies to pass. Band-pass filters (curve (d)) allow only those frequencies within a well-defined band to pass, as shown in Figure 4.7. Although it is possible to construct digital high-pass and band-pass filters, these

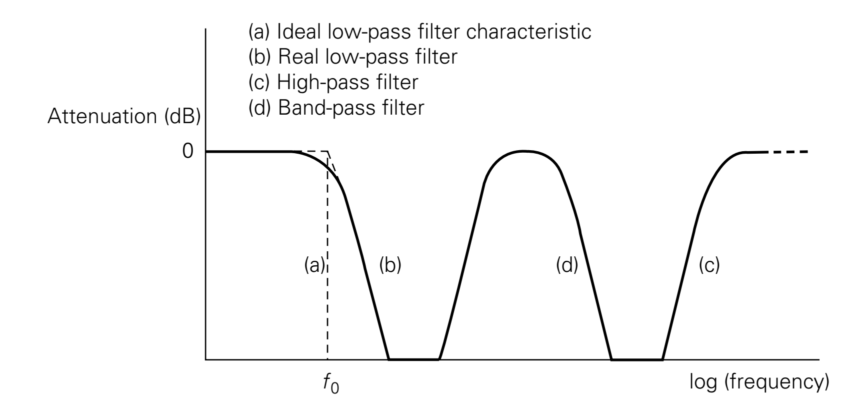


Figure 4.7 *Typical filter characteristics*

are rarely needed for real-time filtration and we will, therefore, concentrate on low-pass filters.

The filter characteristic generally has a rounded shoulder, so the cut-off point is not sharp. The attributes of the filter may be defined by reference to several different points. Sometimes, the frequency at which the signal is attenuated to −3 dB is quoted. In other instances, the curve is characterized by extrapolating the linear, sloping portion of the curve back to the 0 dB level in order to define the cut-off frequency, f_0.

In most situations, the noise suppression properties of a filter are only weakly dependent upon f_0. Small differences in f_0 from some ideal value generally have only a small effect on noise attenuation. This is fortunate as it can sometimes allow a rough approximation to the desired filter characteristic to be used. However, it is always important to carefully assess the dynamic behaviour of digital filter designs to ensure that they operate as expected and within specified tolerances. In particular, when applying a digital filter to an acquired data stream, you should be aware of the effect of the filter's bandwidth on the dynamic performance of the system. It is not only frequencies greater than f_0 that are affected by low-pass filters. The filter characteristic may also significantly attenuate signals whose frequencies are up to an order of magnitude *less* than the cut-off frequency. A signal frequency of $f_0/8$, for example, may be attenuated by typically 0.25 per cent.

Software considerations

When assessing the performance of a digital filter design, the programmer should bear in mind that whatever formulae and algorithms the filter is based on, the actual coded implementation will be subject to a number of potential errors. The ADC quantization and linearity errors will, of course, ultimately limit the accuracy of the system. However, there is another possible source of error which should be considered: the accuracy of the floating-point arithmetic used.

Some filter algorithms are recursive, using the results of previous calculations in each successive iteration. This provides the potential for floating-point rounding errors to accumulate over time. If rounding errors are significant, the filter may become unstable. This can cause oscillations or an uncontrolled rise in output. It may also prevent the filter's output from decaying to zero when the input signal is removed (i.e. set to zero). Filter routines should normally be implemented using high precision arithmetic. Using C's `double` or `long double` types, rather than the `float` data type, will usually be sufficient to avoid significant rounding errors.

Although floating-point software libraries can be employed to perform the necessary calculations, a numeric coprocessor will greatly enhance throughput. The speed of the filter routines may be improved by coding them so as to minimize the number of multiplication and division operations required for each iteration. Where you have to divide a variable by a constant value, multiplying by the inverse of the constant instead will generally provide a slight improvement in execution speed.

Testing digital filters

It is essential that you thoroughly check the performance of all filter routines before you use them in your application. This can be accomplished by creating a test routine or program which generates a series of cosinusoidal signals over a range of different frequencies. At each frequency, f, the signal is given by:

$$s = \cos(2\pi f t) \tag{4.8}$$

where t represents elapsed time. In practice, the signal, s, can be determined at each sample time without recourse to real-time calculations by expressing t as the ratio of the ordinal index, k, of each sample to the sampling frequency, ν, giving

$$s = \cos\left(2\pi k \frac{f}{\nu}\right. \tag{4.9}$$

So, we can generate the signal for a range of different relative frequencies (f/ν). Starting from a maximum value of $\frac{1}{2}$ (the Nyquist limit), the ratio f/ν should be gradually reduced until the desired frequency range has been covered.

For each frequency used, s should be evaluated repeatedly in a loop (with k being incremented on each pass through the loop) and each value of s should be passed to the digital filter routine. The filtered cosinusoidal signal can then be reconstructed and its amplitude and phase determined and plotted against f/ν. Note that the filter's output will generally be based on a history of samples. Because of this the filter will require a certain number of sampled data points before reaching a steady state. You should, therefore, allow sufficient iterations of the loop before assessing the amplitude and phase of the filtered signal.

Simple averaging techniques

The most obvious way of reducing the effects of random noise is to calculate the average of several readings taken in quick succession.

If the noise is truly random and equally distributed about the actual signal level it should tend to average out to zero. This approach is very simple to implement and can be used in applications with fixed signals (e.g. dimensional gauging of cast steel components) or with very slowly varying signals (e.g. temperature measurements within a furnace). If the signal changes significantly during the sampling period, the averaging process will, of course, also tend blur the signal. The period between samples must be short enough to prevent this but also long enough to allow true averaging of low frequency noise components.

The main drawback with the simple averaging process – particularly in continuous monitoring or control systems – is that the filter's output is updated at only $1/N$th of the sampling rate (where N is the number of samples over which the average is calculated). If the filtered signal is then used to generate an analogue control signal, the delay between successive outputs will increase the magnitude of the reconstruction error.

The simple averaging method is useful in a number of situations. However, if it is necessary to measure changing signals in the presence of noise, a more precise analysis of the filter's frequency characteristics are required and it is usually preferable to employ one of the simple low-pass filtering techniques described in the following section.

Low-pass filtering techniques

Ideally a software filter routine should be invoked once for each new sample of data. It should return a filtered value each time it is called, so that the filtered output is updated at the sampling frequency.

There are two distinct classes of filter: recursive and non-recursive. In a non-recursive filter, the output will depend on the current input as well as on previous inputs. The output from recursive filters, on the other hand, is based on previous *output* values and the current input value. The ways in which the various input and output values are combined varies between different filter implementations, but in general each value is multiplied by some constant weight and the results are then summed to obtain the filtered output.

If we denote the sequence of filter outputs by y_k and the inputs (samples) by x_k, where k represents the ordinal index of the iteration, a non-recursive filter is described by the equation:

$$y_k = \sum_{i=0}^{i=k} a_i x_{k-i} \tag{4.10}$$

Here, the constants a_i represent the weight allotted to each element in the summation. In general the series of a_i values is defined so that the most recent data is allocated the greatest weight. The a_i constants often follow an exponential form which allows the filter to model an electronic low-pass filter based on a simple RC network.

The non-recursive filter described by Equation 4.10 is termed an Infinite Impulse Response (IIR) filter because the summation takes place over an unbounded history of filter inputs (i.e. $x_k + x_{k-1} + \cdots + x_2 + x_1 + x_0$). In practice, most non-recursive filter implementations truncate the summation after a finite number of terms, n, and are termed Finite Impulse Response (FIR) filters. In this case, the non-recursive filter equation becomes:

$$y_k = \sum_{i=0}^{i=n-1} a_i x_{k-i} \tag{4.11}$$

Recursive filters are obtained by adding a recursive (or auto-regressive) term to the equation as follows:

$$y_k = \sum_{i=1}^{i=k} b_i y_{k-i} + \sum_{i=0}^{i=k} a_i x_{k-i} \tag{4.12}$$

The constants b_i in the new term represent weights that are applied to the sequence of previous filter *outputs*. Equation 4.12 is, in fact, a general form of the filter equation known as an Auto Regressive Moving Average (ARMA) filter. As we shall see later, this equation can be simplified to form the basis of an effective low-pass recursive filter.

In addition, the following sections cover two implementations of the non-recursive type of filter (the unweighted moving average and the exponentially weighted FIFO). Other filters can be constructed from Equations 4.11 or 4.12, but for most applications one of the three simple filters described below will usually suffice.

Each weight in Equations 4.11 and 4.12 may take either positive or negative values, but the sum of all of the weights must be equal to 1. In a non-recursive filter, the output signal is effectively multiplied by the sum of the weights and if this is not unity the output will be scaled up or down by a fixed factor. The result of using weights which sum to a value greater than 1 in a recursive filter is more problematic. The filter becomes unstable and the output, effectively multiplied by an ever increasing gain, rises continuously.

Equations 4.10 to 4.12 indicate that the time at which each sample, x_k, is obtained is not needed in order to calculate the filter output. It is, therefore, unnecessary to pass time data to the filter routines themselves. However, the rate at which the signal is sampled does,

of course, have a direct bearing on the performance of the filter. For any given set of filter parameters (i.e. a_i, b_i and n), the filter's frequency response curve is determined *solely* by the sampling rate, ν. For example, a filter routine which has a cut-off frequency, f_0, of 10 Hz at $\nu = 100$ Hz will possess an f_0 of 5 Hz if ν is reduced to 50 Hz. For this reason we will refer to the filter's frequency characteristics in terms of the frequency ratio, f/ν (or f_0/ν when referring to the cut-off frequency).

Unweighted moving average filter

The unweighted moving average filter (also sometimes known simply as a moving average filter) is a simple enhancement of the block average technique. It is actually a type of non-recursive filter based on Equation 4.11. The weights a_i are each set equal to $1/n$ so that they sum to unity. The filter is described by the following equation:

$$y_k = \frac{1}{n} \sum_{i=0}^{i=n-1} x_{k-i} \tag{4.13}$$

A FIFO buffer (see Chapter 6) is used to hold the series of x values. The output of the filter is simply the average of all entries held in the FIFO buffer. Because the weights are all equal, this type of filter is also known as an unweighted FIFO filter.

Filters with large FIFO buffers (i.e. large values of n) provide good high-frequency attenuation. They are useful for suppressing noise and unwanted transient signal variations that possess wide-tailed distributions, such as might be present when monitoring the thickness of a rolled sheet product such as rubber or metal sheet.

Listing 4.1 illustrates how the moving average filter can be implemented. The size of the FIFO buffer is determined by the value defined for `N`. The `InitFilter()` function should be called before filtering commences in order to initialize the various FIFO buffer variables. Each subsequent reading `(X)` should be passed to the `Filter()` function which will then return the present value of the moving average.

The filter is, of course, least effective during its start-up phase when part of the FIFO buffer is still empty. In this phase, the filter's output is calculated by averaging over only those samples which have so far been acquired, as illustrated in the listing. `N` calls to the `Filter()` function are required before the FIFO buffer fills with data.

The unweighted moving average filter possesses the frequency characteristic shown in Figure 4.8. It is clear from the figure that larger FIFO buffers provide better attenuation of high frequencies.

Listing 4.1 *An unweighted moving average filter*

```
#define N 100         /* Size of FIFO Buffer */

double  FIFO[N];
int     FIFOPtr;
double  FIFOEntries;
double  FIFOTotal;


void InitFilter()
{
FIFOPtr     = -1;
FIFOEntries = 0;
FIFOTotal   = 0;
}


double Filter(double X)
{
if (FIFOPtr < (N-1))
       FIFOPtr++;
  else FIFOPtr = 0;

if (FIFOEntries < N)
       {
       FIFOTotal     = FIFOTotal + X;
       FIFO[FIFOPtr] = X;
       FIFOEntries   = FIFOEntries + 1;
       }
  else {
       FIFOTotal     = FIFOTotal - FIFO[FIFOPtr] + X;
       FIFO[FIFOPtr] = X;
       }

return FIFOTotal / FIFOEntries;
}
```

However, because of resonances occurring at even values of ν/f and where the FIFO buffer contains an integer number of signal cycles (i.e. when nf/ν is an integer), oscillations are present in the characteristic curve at frequencies higher than f_0. As a rough rule-of-thumb, the cut-off frequency is given by $\nu/f_0 \sim 2.5n$ to $3.0n$.

As with all types of filter, a phase lag is introduced between the input and output signals. This tends to increase at higher frequencies. Because of the discrete nature of the sampling process and the resonances described above, the phase vs. frequency relationship also becomes irregular above the cut-off frequency.

This type of filter is very simple, but is ideal in applications where high speed filtration is required. If there is a linear relationship

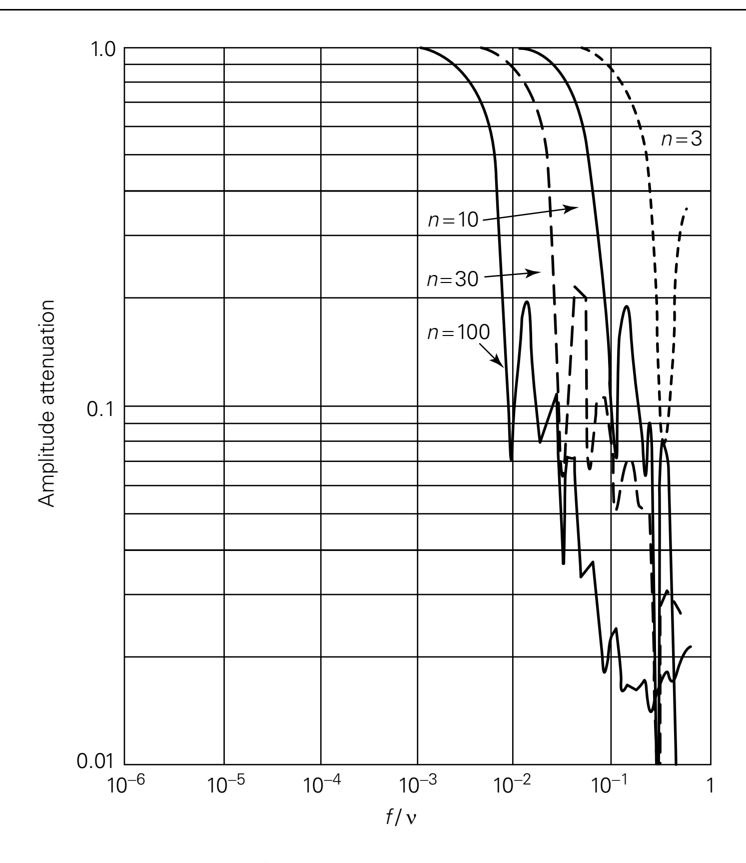


Figure 4.8 *Attenuation vs. frequency relationship for the unweighted moving average filter*

between the measurand and the corresponding digitized reading, the unscaled ADC readings can be processed directly using a moving average filter based on simple integer (rather than floating-point) arithmetic.

Exponentially weighted FIFO filter

The unweighted moving average filter gives equal weight to all entries in the FIFO buffer. Consequently, a particularly large reading will not only affect the filter output when it is supplied as a new input, it will also cause a large change in output when the reading passes through the FIFO buffer and is removed from the summation. To minimize the latter effect, we may apply a decreasing weight to the readings as they pass through the buffer so that less attention is paid to older entries. One such scheme employs an exponentially decreasing series

of weights. In this case the weights a_i in Equation 4.11 are given by:

$$a_i = e^{-(it/\tau)} \tag{4.14}$$

Here, t represents the time interval between successive samples (equal to $1/\nu$) and τ is the time constant of the exponential filter-response function. In an ideal filter, with a sufficiently large FIFO buffer, the series of exponential weights will not be truncated until the weights become insignificantly small. In this case the time constant, τ, will be related to the desired cut-off frequency by:

$$f_0 = \frac{1}{2\pi\tau} \tag{4.15}$$

Obviously, in a real filter, the finite size of the FIFO buffer will modify the frequency response, but this effect will be small provided that $nt \gg \tau$.

For the purpose of calculating the weights, it is convenient to make use of a constant, r, which represents the number of characteristic exponential time periods (of length τ) that are encompassed by the FIFO buffer:

$$r = \frac{nt}{\tau} \tag{4.16}$$

The weights are then calculated from:

$$a_i = e^{-(ir/n)} \tag{4.17}$$

Substituting Equation 4.16 into Equation 4.15 (and remembering that $t = 1/\nu$) we see that the expected cut-off frequency of the filter is given by:

$$\frac{f_0}{\nu} = \frac{1}{2\pi} \cdot \frac{r}{n} \tag{4.18}$$

This applies only for large values of r (i.e. greater than about 3 in practice) which allow the exponential series of weights to fall from unity – for the most recent sample – to a reasonably low level (typically <0.05) for the oldest sample. Smaller values of r give more weight to older data and result in the finite size of the FIFO buffer becoming the dominant factor affecting the filter's response.

Listing 4.1 may be easily adapted to include a series of exponential weights as illustrated in Listing 4.2. The `InitFilter()` function, which must be called before filtering commences, first calculates a `WeightStep` value equivalent to the ratio of any two adjacent weights: a_i/a_{i-1}. It also determines the sum of all of the weights. This is

Listing 4.2 *An exponentially weighted FIFO filter*

```
#define N 100            /* Size of the FIFO buffer */
#define R 3              /* No. of characteristic time periods within buffer */

double  WeightStep;
double  SumWeights;
double  LowWeight;
double  FIFO[N];
int     FIFOPtr;
double  FIFOEntries;
double  FIFOTotal;


void InitFilter()
{
double T;
double Weight;
int    I;

T = R;
WeightStep = exp(-1 * T / N);
SumWeights = 0;
Weight     = 1;
for (I = 0; I < N; I++)
  {
  Weight     = Weight * WeightStep;
  SumWeights = SumWeights + Weight;
  }
LowWeight = Weight;

FIFOPtr     = -1;
FIFOEntries = 0;
FIFOTotal   = 0;
}


double Filter(double S)
{
if (FIFOPtr < (N-1))
       FIFOPtr++;
  else FIFOPtr = 0;

if (FIFOEntries < N)
       {
       FIFOTotal     = (FIFOTotal + S) * WeightStep;
       FIFO[FIFOPtr] = S;
       FIFOEntries   = FIFOEntries + 1;
       }
  else {
       FIFOTotal     = (FIFOTotal - (FIFO[FIFOPtr] * LowWeight) + S) * WeightStep;
       FIFO[FIFOPtr] = S;
       }

return FIFOTotal / SumWeights;
}
```

required for normalizing the filter output. `LowWeight` is the weight applied to the oldest entry in the FIFO buffer and is needed in order to calculate the affect of removing the oldest term from the weighted total.

The `Filter()` function should be called for each successive sample. This function records the `N` most recent samples (i.e. `X` values) in a FIFO buffer. It also maintains a weighted running total of the FIFO contents in `FIFOTotal`. The weights applied to each entry in the buffer are effectively reduced by the appropriate amount (by multiplying by `WeightStep`) as each new sample is added to the buffer.

Good high frequency attenuation is obtained with $r > 1$, particularly with the larger FIFO buffers. Phase shifts similar to those described for the moving average filter also occur with the exponentially weighted FIFO filter. Again the effects of resonances and discrete sampling introduce irregularities in the attenuation and phase vs. frequency relationships. As would be expected, this effect is more prominent with values of *r* less than about 1 to 3. The cut-off frequencies obtained with various combinations of *r* and *n* are shown in Figure 4.9.

When *r* is greater than about 3, the f_0/ν data agrees closely with the expected relationship (Equation 4.18). Slight deviations from the ideal response curve are due to the discrete nature of the sampling. Values of *r* less than about 3 result in a somewhat higher cut-off frequency for a given value of r/n. Conversely, increasing *n* will reduce f_0.

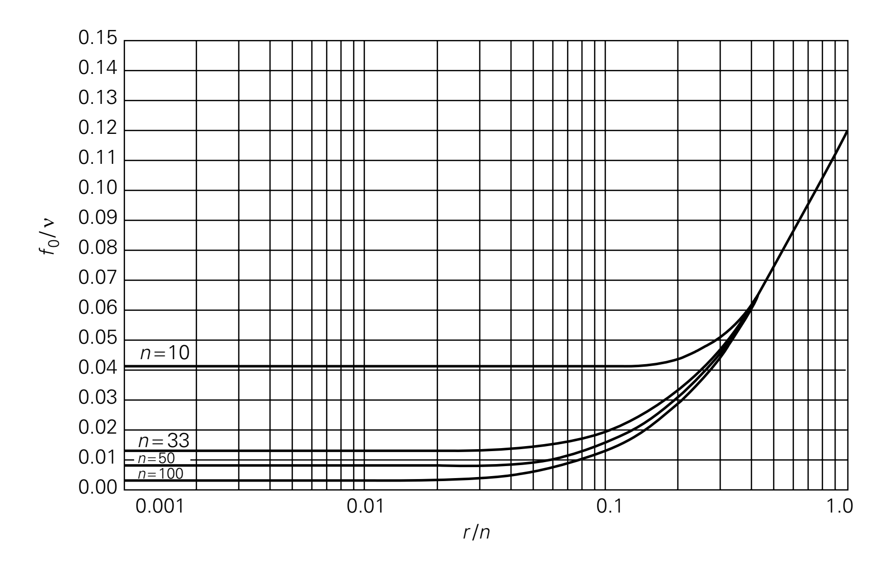


Figure 4.9 *Cut-off frequencies vs. r/n for exponentially weighted FIFO filters*

The data in Figure 4.9 is replotted in Figure 4.10 which may be used as a basis for choosing values of r and n in practical applications. To determine the values of n and r that are necessary to obtain a given f_0:

1. Determine ν (remembering that it should be high enough to avoid aliasing) and then calculate the desired f_0/ν.

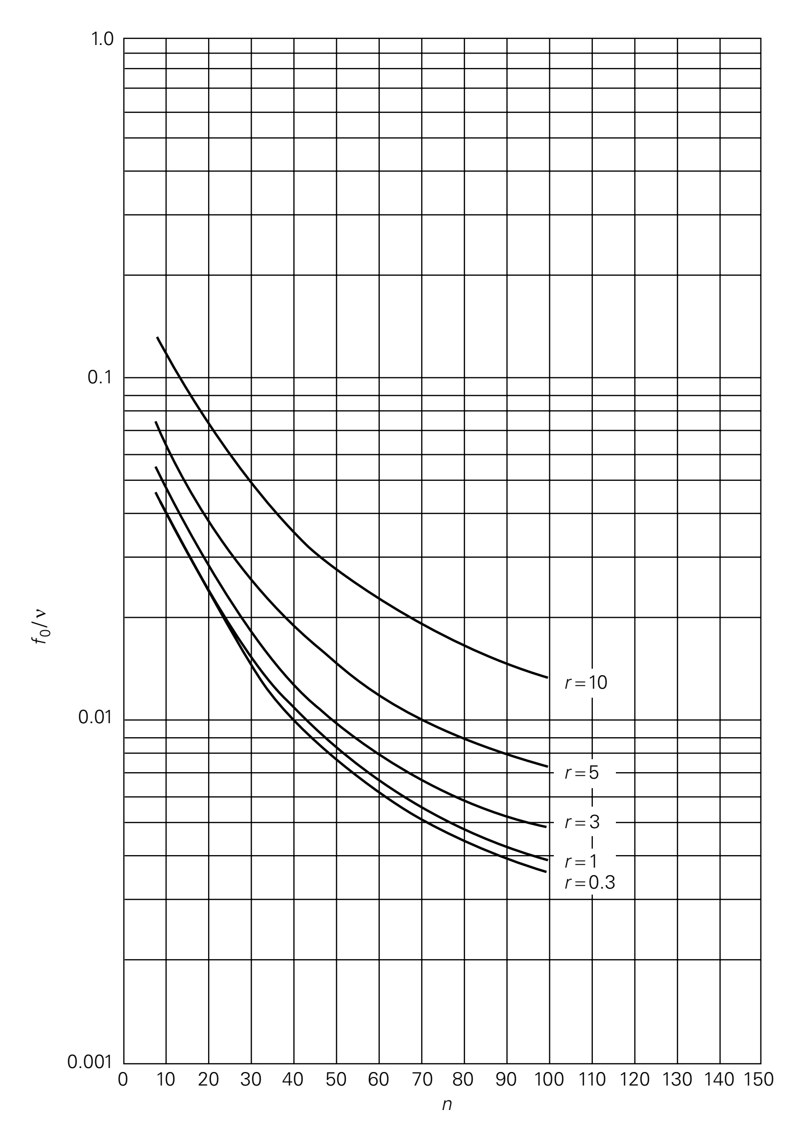


Figure 4.10 *Cut-off frequencies vs. n for exponentially weighted FIFO filters*

2. Refer to Figure 4.10 to choose a suitable combination of r and n. The optimum value of r is generally about 3, but values between about 1 and 10 can give adequate results (depending upon n).
3. Consider whether the FIFO buffer size (n) indicated is practicable in terms of memory requirements and filter start-up time. If necessary use a smaller FIFO buffer (i.e. smaller n) and lower value of r to achieve the desired f_0.

A number of points should be borne in mind when selecting r and n. With small r values, a greater weight is allocated to older data and this lowers the cut-off frequency.

When $r < 1$ the filter behaves very much like an unweighted moving average filter because all elements of the FIFO buffer have very similar weights. The cut-off frequency is then dependent only on n (i.e. it is only weakly dependent on r) and is determined by the approximate relationship $f_0/\nu \sim (2.5n)^{-1}$ to $(3n)^{-1}$. Only when r is greater than about 2 to 3 is there any strong dependence of f_0 on r.

When r is greater than about $n/3$, the performance of the filter depends only on the ratio r/n because the exponential weights fall to an insignificantly small level well within the bounds of the FIFO buffer. There is usually no advantage to be gained from operating the filter in this condition as only a small portion of the FIFO buffer will make any significant contribution to the filter's output. If you need to achieve a high f_0 it is far better to increase ν or, if this is not possible, to reduce n, rather than increasing r beyond $n/3$. Best results are often obtained with an r value of about 3. This tends to generate a smoothly falling frequency response curve with a well defined f_0 and good high frequency attenuation.

Recursive low-pass filter

A very effective low-pass filter can be implemented using the general recursive filter Equation (4.12). The equation may be simplified by using only the most recent sample x_k (by setting $a_i = 0$ for $i > 0$) and the previous filter output y_{k-1} (by setting $b_i = 0$ for $i \neq 1$). The filter equation then reduces to

$$y_k = ax_k + by_{k-1} \tag{4.19}$$

$$\text{where} \quad a + b = 1 \tag{4.20}$$

In Equation 4.19 the 0 and 1 subscripts have been dropped from the weights a and b respectively. As discussed previously, the condition 4.20 is required for stability. It should be clear that the filter output will respond more readily to changes in x when a is relatively large. Thus the cut-off frequency, f_0, will increase with a. Knowing the

Listing 4.3 *A recursive low-pass filter*

```
#define A 0.1              /* Modify this value as necessary */
double Y;
double B;

void InitFilter()
{
Y = 0;
B = 1.0 - A;
}

double Filter(double X)
{
Y = X * A + Y * B;
return Y;
}
```

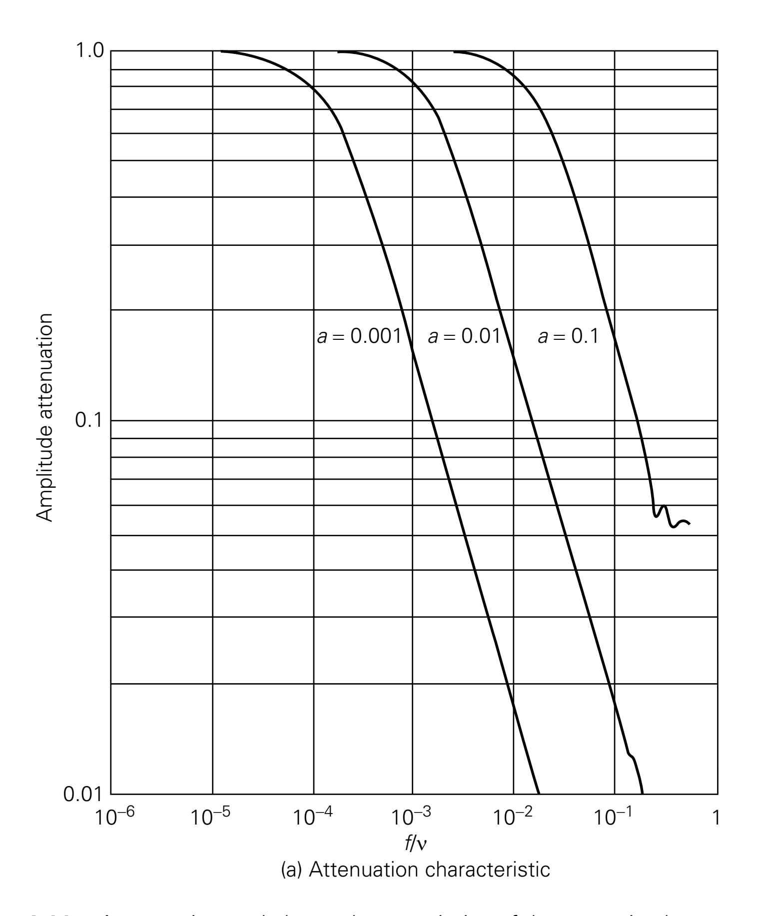


Figure 4.11 *Attenuation and phase characteristics of the recursive low-pass filter*

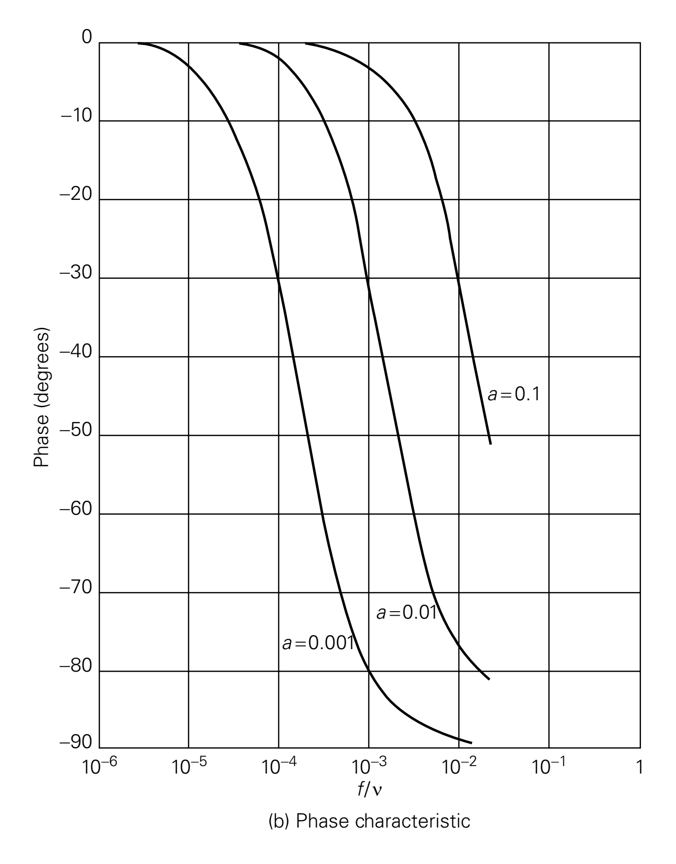
sampling frequency, ν, the constant a can be calculated from the required value of f_0 as follows:

$$a = \frac{\dfrac{2\pi f_0}{\nu}}{\dfrac{2\pi f_0}{\nu} + e^{-(2\pi f_0/\nu)}} \tag{4.21}$$

When $\nu \gg f_0$, the denominator tends to unity and Equation 4.21 becomes

$$a \approx \frac{2\pi f_0}{\nu} \tag{4.22}$$

Ideally, the cut-off frequency should be somewhat less than $\nu/20$ in order to achieve reasonable attenuation at high frequencies. In this case, the approximation given in Equation 4.22 introduces only a



(b) Phase characteristic

Figure 4.11 *(continued)*

small error in the cut-off frequency and this generally has a negligible effect on the performance of the filter.

Listing 4.3 shows how this simple recursive filter can be implemented in practice. The filter coefficient, a, is defined in the listing as the constant `A`. In this case it is set to 0.1, but other values may be used as required. The `InitFilter()` function must be called before the sampling sequence starts. It initializes a record of the previous filter output, `Y`, and calculates the other filter coefficient, b, which is represented by the variable `B` in the listing. This function may be modified if required to calculate coefficients a and b (i.e. the program variables `A` and `B`) from values of f_0 and ν supplied in the argument list. The `Filter()` function itself simply calculates a new filter output (using Equation 4.19) each time that it is called.

Figure 4.11 illustrates the attenuation and phase lag vs. frequency characteristics obtained with a number of different values of a. The relationship between f_0 and a follows the form expressed in Equation 4.22 very closely. For a given value of f_0/ν, there is little difference between the characteristics of the recursive low-pass filter and the optimum ($r = 3$) exponentially weighted non-recursive (FIFO) filter. In general, however, the recursive filter exhibits a smoother fall-off of response and there are no resonances at high frequencies. The phase vs. frequency curve is also more regular than that obtained with the exponentially weighted FIFO filter. Note that at the cut-off frequency the phase lag is 45°.

This Page Intentionally Left Blank

Part 3 I/O Techniques and Buses

This Page Intentionally Left Blank

5 The interrupt system

The PC's interrupt system provides a means of temporarily suspending (or interrupting) the normal execution of a program in order to allow the processor to respond to specific events. These events may occur either as a result of executing certain instruction sequences or when a peripheral device wishes to request service (e.g. when the keyboard signals that a key has been pressed). The interrupt system is particularly useful in DA&C applications. Interrupts permit the system to react quickly to a variety of control and status inputs. They also allow a degree of synchronism to be maintained between external events and the software routines that are needed to respond to them.

When an interrupt event occurs, the processor usually responds, at the earliest opportunity, by saving its flags register and the address of the next instruction it would otherwise have executed, and then jumping to an *interrupt handler* routine located at a predefined address in memory. In the case of a multitasking operating system, additional, task-related context information is also stored before the interrupt handler is invoked. The interrupt handler performs whatever action is necessary (e.g. reading a key code from the keyboard or digitized data from an ADC) and then returns control of the system to the original process at the point that it was interrupted. In this way, the code contained within an interrupt handler can be executed on demand, providing timely software service for a variety of events or error conditions.

This chapter describes the PC's interrupt system in some detail and illustrates software techniques for creating interrupt handlers for use in data acquisition. It also discusses some important interrupt-related considerations which you should bear in mind when writing data-acquisition software for the PC.

If you are an application developer, rather than an system-level programmer, it is likely that you will need to write interrupt-handling

code only if programming in real mode, for example under MS-DOS or a real-time version of DOS. In 32-bit protected-mode operating systems, such as Windows NT, interrupt handling can be performed only by highly privileged code – i.e. by operating-system code or kernel-mode device drivers. Often, DA&C card manufacturers will provide suitable Windows NT drivers, obviating the need to write your own interrupt code. For this reason, and in order to convey the principles of the topic without unnecessary complication, most of the material in this chapter is presented in the context of a *real-mode* application. Some examples will require adaptation in order to operate under Windows NT and other protected operating systems. Unfortunately a full discussion of protected-mode interrupts and kernel-mode (Ring 0) drivers is beyond the scope of this book. However, a large proportion of this chapter also applies to protected-mode environments, and important differences, such as interrupt response times (latency), are discussed. For further information on Windows device drivers and interrupt handling, refer to the text by Solomon (1998). Buchanan (1999) also provides useful examples of interrupt processing.

The PC supports four different types of control-transfer mechanisms that are all loosely referred to as interrupts: the Non-Maskable Interrupt (NMI), external interrupts, software interrupts and processor exceptions. The nature of the various interrupt mechanisms and the ways in which the interrupts are initiated differ considerably. Software interrupts and external hardware interrupts are usually of most relevance to DA&C applications programs, but you should also be aware of the NMI and processor exception mechanisms, particularly if you are involved in producing time-critical applications or systems software. These topics are discussed in more detail later in this chapter, but first, we will consider the mechanism by which control is transferred to the interrupt handler.

5.1 Interrupt vectors

Whenever any type of interrupt occurs, the processor must transfer control of the system to a suitable interrupt handler. In order for the processor to determine where to jump, it must retrieve the address of the interrupt handler from a table located at a known position in memory. Each address in this table is known as an interrupt vector and consists of 4 bytes which hold the offset (IP) and segment (CS) portions of the address in the standard Intel low–high format. In real mode, the interrupt vector table (IVT) is located at the bottom of addressable memory (i.e. at location 0000:0000h). It is 1024 bytes

long and may contain up to 256 separate interrupt vectors. The PC system can, therefore, accommodate up to 256 different types of interrupt. Some of these are assigned for use by the NMI, external hardware interrupts and exceptions, but the majority are used for software interrupts.

Not all interrupt vectors point to (i.e. contain the address of) executable code. Depending upon the configuration of the system and the software installed, certain interrupt vectors may be configured to point to tables of data etc. Table 5.1 lists the standard interrupt vector usage on the PC.

The BIOS possesses an Unexpected Interrupt Handler routine. All unused hardware interrupts, user interrupts (int 1Ch and 4Ah) and most processor exceptions with Interrupt Type Codes less than 8 are directed to this handler by the BIOS POST routines. If one of these interrupts occur before the operating system or an application has installed a suitable handler, the Unexpected Interrupt Handler is invoked. This immediately sets the Carry Flag and returns control to the interrupted process, preserving all other registers. The Unexpected Interrupt Handler also maintains a record of the last unexpected *external hardware* interrupt at offset 6Bh in the BIOS data area. A single bit in this location is set to denote the IRQ level of the interrupt. For example, an unexpected IRQ5 (interrupt type code 13) would cause the BIOS to store the value 00100000b. Similarly, for an unexpected IRQ7 (type code 15), the value 1000000b would be stored. On the IBM AT and subsequent systems, the IRQ2 bit is set when an unexpected interrupt is detected on IRQ8–IRQ15. External hardware interrupts and IRQ levels are discussed in the following section.

Table 5.1 is by no means a comprehensive list of interrupt usage on the PC. Although most BIOS and DOS interrupts are used consistently throughout the range of PC 'compatible' computers on the market, some of the interrupt vectors may be allocated differently in specific PC systems. The applications and systems software as well as add-in hardware (e.g. network adaptors) present on individual machines will also determine which interrupts are in use. In particular, some of the interrupts in the ranges 50h to 5Fh, 68h to 6Fh, 78h to 7Fh, 88h to B8h and F8h to FFh may be set aside for specific purposes (e.g. relocating hardware interrupts when operating systems software such as Windows, OS/2 or DESQview are installed). Networked systems may also make use of several of the interrupts listed in Table 5.1.

There are already many thousands of software products on the market, all of which need to take advantage of the PC's interrupt system. New products continually come onto the market and these

Table 5.1 *Standard interrupt vector assignments on the IBM PC and compatible machines*

Type	*Description*
00h	Divide-by-zero exception.
01h	Single-step trap (generated after each instruction if TF = 1).
02h	NMI.
03h	Breakpoint (generated by breakpoint opcode CCh).
04h	Overflow (generated by INTO instruction if OF has been previously set).
05h	Print screen.
06h	Reserved.
07h	Reserved.
08h	IRQ0: System timer tick.
09h	IRQ1: Keyboard data available.
0Ah	IRQ2: LPT2 on PC. Reserved on XT. Cascade to slave PIC on AT & PS/2.
0Bh	IRQ3: COM2 or COM4.
0Ch	IRQ4: COM1 or COM3.
0Dh	IRQ5: Fixed disk on PC, XT. LPT2 on AT. Reserved on PS/2.
0Eh	IRQ6: Diskette controller.
0Fh	IRQ7: LPT1.
10h	BIOS video services.
11h	BIOS equipment-check service.
12h	BIOS memory size service.
13h	BIOS diskette I/O service.
14h	BIOS communications service.
15h	BIOS miscellaneous services.
16h	BIOS keyboard services.
17h	BIOS printer services.
18h	BIOS ROM BASIC entry point.
19h	BIOS bootstrap loader.
1Ah	BIOS time-of-day services.
1Bh	Ctrl-Break handler.
1Ch	Timer tick user interrupt (invoked from int 08h).
1Dh	Pointer to BIOS's video parameter table. Not an interrupt vector.
1Eh	Pointer to BIOS's diskette parameter table. Not an interrupt vector.
1Fh	Pointer to BIOS's 8 × 8 graphics font. Not an interrupt vector.
20h	DOS program termination. Now obsolete, but retained for compatibility.
21h	DOS services.
22h	DOS program termination routine. Not an interrupt vector.
23h	DOS Ctrl-C/Break handler. Invoked when DOS detects Ctrl-C or Ctrl-Break.
24h	DOS critical error handler.

Table 5.1 *(continued)*

Type	*Description*
25h	DOS absolute disk read service.
26h	DOS absolute disk write service.
27h	DOS terminate and stay resident service.
28h	DOS idle interrupt.
29h	DOS fast console character output.
2Ah–2Dh	Reserved.
2Eh	DOS command interpreter interface.
2Fh	DOS multiplex interrupt.
30h	Reserved.
31h	DPMI programming interface.
32h	Reserved. Infrequently used.
33h	Mouse driver services.
34h–3Eh	Floating-point emulation in Microsoft and Borland programming languages.
3Fh	Overlay and DLL management in Microsoft and Borland languages.
40h	BIOS diskette I/O (interrupt 13h revectored by hard disk BIOS).
41h	Pointer to BIOS's hard disk #0 parameter table. Not an interrupt vector.
42h	BIOS default video services (revectored from int 10h by EGA/VGA BIOS).
43h	Pointer to BIOS's graphics character table. Not an interrupt vector.
44h	Pointer to PCjr BIOS's graphics character table. Not an interrupt vector.
45h	Reserved. Infrequently used.
46h	Pointer to BIOS's hard disk #1 parameter table. Not an interrupt vector.
47h	Reserved. Infrequently used.
48h	Keyboard on PCjr. Reserved on all other systems.
49h	Keyboard on PCjr. Reserved on all other systems.
4Ah	BIOS real-time clock user alarm interrupt.
4Bh	SCSI device interface. Virtual DMA services.
4Ch	Reserved. Infrequently used.
4Dh	Reserved. Infrequently used.
4Eh	Reserved. Infrequently used.
4Fh	SCSI device interface.
50h–5Fh	Reserved. Some vectors used by DESQview, OS/2, Windows 95 and networks.
60h–66h	User interrupts.
67h	LIM EMS and VCPI.
68h–6Fh	Reserved. Some vectors used by network products.
70h	IRQ8: Real-time clock periodic/alarm interrupt. AT and PS/2.
71h	IRQ9: Reserved. Invoked via IRQ2 bus line. AT and PS/2.

continued overleaf

Table 5.1 *(continued)*

Type	*Description*
72h	IRQ10: Reserved.
73h	IRQ11: Reserved.
74h	IRQ12: Pointing device interrupt (e.g. PS/2 mouse). PS/2 and AT compatibles.
75h	IRQ13: Numeric coprocessor. AT and PS/2.
76h	IRQ14: Hard disk controller. AT and PS/2.
77h	IRQ15: Reserved.
78h–7Fh	Reserved. Some vectors used for network products.
80h–85h	Reserved for BASIC.
86h–EEh	IBM ROM BASIC interpreter. Some vectors also used by network products.
EFh–F0h	IBM ROM BASIC interpreter. Compiled BASIC.
F1h–FDh	User interrupt on AT and PS/2. Reserved on PC and XT.
FEh	Reserved.
FFh	Reserved.

also require new interrupts to be assigned. As there are only 256 available interrupt vectors, a degree of overlap is sometimes inevitable. Fortunately, many software packages and hardware products (e.g. data-acquisition cards) help to avoid interrupt conflicts by allowing the user some latitude in selecting which interrupts are to be used.

For these reasons, published interrupt tables tend to differ slightly, often listing many of the interrupts simply as ‘Reserved’ and, in general, it is wise to avoid using any of these in your own software.

One must also bear in mind that there can in some circumstances be ambiguity over the usage of a specific interrupt vector. Several of the first 32 vectors are used on the PC for processor exceptions as well as for external hardware interrupts or BIOS services. This overlap arises from the design of the original PC and has become more problematic as new processor features and exceptions have been introduced. Contentions tend not to arise when the processor is running in real mode, but protected mode software must ensure that it can identify the source of an interrupt unambiguously. The full implications of interrupt conflicts and techniques to resolve them are beyond the scope of this book. However, such considerations are usually handled by protected-mode operating systems. Windows 95 and DESQview, for example, avoid such problems by remapping hardware interrupts to different vectors. Further details of interrupt conflicts and the interrupt relocation technique may be found in the text by van Gilluwe (1994).

Brown and Kyle (1991) provide a thorough and detailed account of interrupt usage on the PC. This publication includes a great deal of information on the interrupts used by specific software and hardware products, and it is recommended that this text should be consulted whenever you need to select interrupts to be used in a data-acquisition system. This should help to achieve compatibility with other products by avoiding any interrupts which they might use. However, if you are concerned only with picking a suitable external hardware interrupt (IRQ) for interfacing to a data-acquisition card, for example, the choice is usually much simpler and the tables provided in Appendix A should assist in these circumstances.

5.2 Hardware interrupts

The NMI and external interrupts are, in fact, both types of hardware interrupt. The processor is equipped with two pins known as NMI and INTR. Signals present on either of these pins can interrupt the processor. The INTR line carries external hardware interrupt requests, while the NMI line carries non-maskable interrupt requests. In the PC, a number of different subsystems and peripheral components are able to assert the NMI or INTR lines whenever they require attention from the processor.

External hardware interrupts

External interrupt requests may occur at any time during execution of a program. Because they are asynchronous with the operation of the processor, the programmer should make no assumptions about when an interrupt might be generated. As an interrupt handler may take control of the system for perhaps a few hundred microseconds at a time (or more in some cases), the possibility of an interrupt occurring can clearly affect the ability of non-interrupt code to operate in accordance with the tight timing constraints that are often required of DA&C systems. It is sometimes preferable to place time-critical code inside interrupt handlers, as this can help to ensure that the system responds to external stimuli within predefined time limits. However, as we shall see, it is not always easy to achieve a guaranteed response time, even with interrupts.

There are other problems inherent in using an asynchronous interrupt system. The interrupt handler may have to read or modify global data structures or to access hardware resources. It is clearly important to prevent interrupt routines and non-interrupt code from accessing shared resources (such as global data and hardware)

at the same time. Suppose that the non-interrupt portion of your program begins to execute a sequence of instructions which reads 16 bytes from a global array. If an interrupt occurs before the reading sequence is completed and the interrupt handler changes the contents of the array, the non-interrupt code will, when it regains control, read the modified data from the remainder of the array. There will consequently be a mismatch between the first and last bytes read from the array. Similar and sometimes more catastrophic consequences may result if the shared resource in question is a critical item of hardware.

It is possible to circumvent these problems to some extent by temporarily disabling the external interrupt system. The processor can be programmed to *mask* external hardware interrupts by means of the `CLI` (Clear Interrupt Flag) assembly language instruction. This resets the processor's Interrupt Flag (IF) causing the processor to ignore any external hardware interrupt requests that it receives on the INTR line. By this means it is possible to prevent interrupts from occurring and thereby to protect critical portions of the code. At the end of the critical section, interrupts may be enabled again by issuing the `STI` instruction which sets IF back to 1. If you disable external interrupts in this way, do not keep them disabled for too long as this will affect the speed at which other interrupt driven processes can respond. Try to confine the critical code to just a few machine instructions if possible. This helps to ensure that all interrupts are serviced in a timely manner.

Note that none of the other interrupts (i.e. NMI, processor exceptions or software interrupts) can be masked in this way although, as we shall see later in this chapter, the design of the PC does provide a mechanism for controlling whether NMI signals reach the processor.

Introduction to the 8259A PIC

The external hardware interrupt system was managed on the original IBM PC and XT machines by an Intel 8259A Programmable Interrupt Controller (PIC) as shown in Figure 5.1. The INTR line can be asserted by the PIC whenever it receives an interrupt request signal from one of eight peripheral devices. A similar system was adopted for the IBM AT, but in this machine a second 8259A PIC was added to provide seven further interrupt request (IRQ) lines. Most modern ISA and EISA PCs provide the same dual-PIC functionality using compatible custom circuitry. As this arrangement is functionally equivalent, we will refer only to the 8259A PICs in the remainder of this chapter.

All but two IRQ lines are made available to expansion cards on the ISA/EISA bus. The PCI bus present in most modern PCs carries

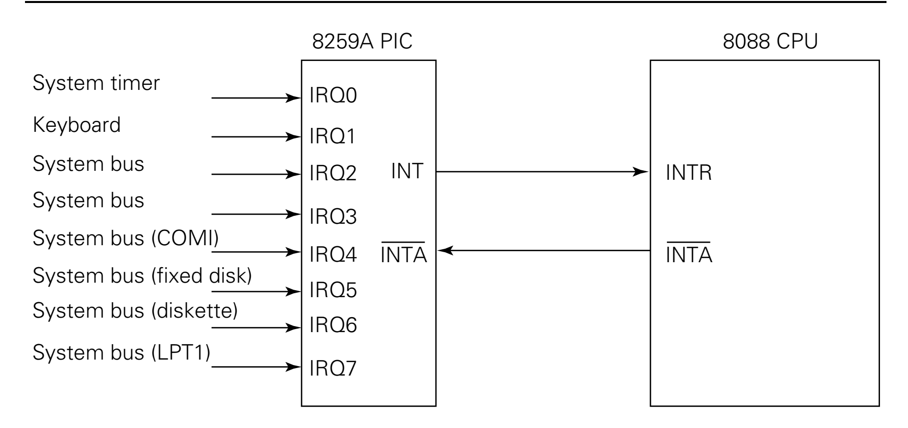


Figure 5.1 *The IBM XT's external hardware interrupt system*

four separate interrupt request lines, and these are automatically mapped by the PCI–ISA bridge hardware to one of the PIC's IRQ lines (i.e. IRQ3–IRQ7, IRQ9–IRQ12, IRQ14 or IRQ15).

As its name suggests, the PIC is a programmable device which may be made to operate in a variety of different modes. It is preprogrammed to a default operating mode by the BIOS's start-up code. Most applications make use of this default configuration, but a few more specialized systems reprogram the PIC. Unless stated otherwise, the remainder of this section will discuss how the PIC functions in its default operating mode.

When two or more interrupt conditions occur at the same time, the system must decide which interrupt request it will respond to first. The processor prioritizes the various types of interrupt and, in normal operation, gives all INTR requests (i.e. external hardware interrupts) the lowest priority. The principal function of the PIC is to prioritize these external hardware interrupt requests (IRQ) signals and to issue a corresponding sequence of INTR signals to the processor. The default operating mode assigns highest priority to IRQ0 and the lowest priority to IRQ7. A similar sequence applies to the secondary PIC present on the AT and compatible machines although the highest and lowest priority interrupt lines are in this case referred to as IRQ8 and IRQ15 respectively. This priority scheme means that an interrupt handler may itself be interrupted by a higher priority interrupt request (provided that the processor's Interrupt Flag is set), but lower priority requests must wait until the present interrupt level has been cleared.

The PIC incorporates several 8-bit registers which are used for manipulating the interrupt request signals as shown in Figure 5.2. The interrupt request signals are latched in the Interrupt Request

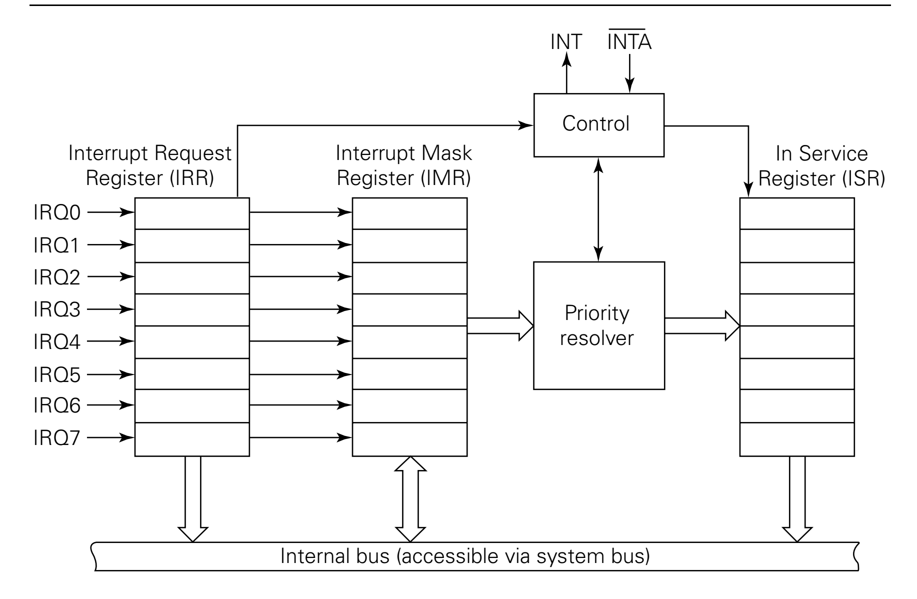


Figure 5.2 *Schematic diagram of the main elements of the 8259A PIC*

Register (IRR). The IRR may be programmed to record either edge-triggered or level-triggered interrupt signals. The trigger method used is dependent upon the type of machine and should not normally be changed by the programmer. The latched IRR signals are then passed to the Interrupt Mask Register (IMR) which contains a user-programmable bit pattern that selectively enables or disables interrupt requests on certain IRQ lines. A low bit placed in this register will enable the associated interrupt. Next, the interrupt signals are then passed collectively to the priority resolver which prioritizes all pending (and enabled) requests. The result of this operation is that the INT line (which is connected to the processor's INTR line) is asserted and this initiates the interrupt sequence. In addition, 1 bit of the In Service Register (ISR) is set to indicate which of the pending interrupts is currently being serviced.

The IRR, IMR and ISR may be read by software in order to determine the current state of the interrupt system. As already mentioned, the software can also write to the IMR to selectively enable or disable certain IRQ lines. Each bit in the IMR corresponds to one IRQ line and has no effect on any higher or lower priority lines. The PIC also incorporates a number of other registers which allow the operating mode of the device to be programmed.

Many plug-in adaptor cards provide jumpers or DIP switches for the purpose of selecting which IRQ line (if any) the card is to use. It is, of course, important to ensure that no two devices are assigned to

the same IRQ line unless you are able to make use of the interrupt sharing facilities that exist on MCA and EISA machines. Table A.2 (in Appendix A) lists the standard IRQ assignments used on the PC. Remember that the actual assignments may vary between individual computers, so it is wise to keep a record of which IRQ lines are utilized by each adaptor card in the system.

The interrupt sequence

When an adaptor card asserts one of the IRQ lines, it sets in motion the following series of events which ends in the execution of an associated interrupt handler routine.

1. When the peripheral device requires the processor's attention, it drives its allotted IRQ line high (on the PCI bus the interrupt request signal is steered by bridge hardware to the appropriate IRQ line).
2. The IRQ signal is latched into the PIC's IRR (this is either edge or level triggered, depending upon the class of PC in use) and if the corresponding bit of the IMR is clear, the interrupt request is passed (with any other pending requests) to the priority resolver.
3. If no higher priority interrupts are pending, the PIC initiates the interrupt by asserting the processor's INTR line. If a higher priority interrupt is pending or currently in service, the PIC will wait until all higher priority interrupts have been serviced before proceeding with the new interrupt request.
4. When the processor receives the INTR signal from the PIC it asserts the PIC's Interrupt Acknowledge ($\overline{\text{INTA}}$) line twice in succession. The processor waits until it has completed the current instruction before acknowledging the interrupt. If external interrupts have been disabled (IF = 0), the processor will not acknowledge the interrupt and the $\overline{\text{INTA}}$ line is not asserted. The PIC responds to the first INTA cycle by setting the appropriate bit of the ISR and clearing the corresponding IRR bit. The second INTA cycle causes the PIC to transfer an 8-bit Interrupt Type Code (the zero-based ordinal index of the interrupt vector to be used) to the processor via the data bus. The value of this code depends upon the IRQ line which generated the interrupt and also upon how the PIC has been initialized (see *Remapping interrupts* later in this chapter).
5. The processor retrieves the Interrupt Type Code from the data bus and multiplies it by four to calculate the offset into the IVT of the interrupt vector that it will use.
6. The processor saves its Flags register on the stack and then clears its Interrupt and Trap flags. At this point, the segment

and offset addresses of the next instruction that the processor would otherwise have executed are also pushed onto the stack (these are used to resume execution of the interrupted code when the interrupt handler terminates). The processor retrieves the address of the interrupt handler from the interrupt vector and, by placing this address into its CS:IP registers, effectively transfers control to the beginning of the interrupt handler.

7. The interrupt handler performs whatever actions are necessary in order to respond to the peripheral device's interrupt request. These actions will vary, but should always result in the device removing its request by pulling the appropriate IRQ line low again. Before returning control to the interrupted process, the handler should then issue an End Of Interrupt (EOI) command (usually a value of 20h) to the PIC. The EOI command causes the ISR to be reset, allowing further interrupt requests of an equal or lower priority to proceed. The interrupt handler should ensure that it saves the contents of all of the processor's registers and that it restores them before returning. The return itself should be implemented with the `IRET` (Interrupt Return) instruction rather than the normal subroutine return, `RET`. The `IRET` instruction automatically restores the Flags register (and therefore the Interrupt Flag) which had originally been saved by the processor on the stack. It also loads the return address from the stack into the CS:IP registers to effect the return.

Figure 5.3 illustrates this sequence diagrammatically. The circled numbers refer to the stages in the foregoing list. Bear in mind that this figure is not a precise timing diagram – indeed the timing of certain elements can vary considerably – nor does it include all of the control signals that are passed between the PIC and the processor.

The interrupt sequence in protected mode (e.g. under Microsoft Windows) is similar in many respects, although there are a number of important differences. See Hummel (1992) for more on protected-mode interrupts.

Interrupt triggering

There are two ways in which signals present on the various IRQ lines may become latched into the PIC's IRR and thereby generate an interrupt request: edge-triggered or level-triggered detection. The former method uses the rising edge of the IRQ line to latch the corresponding IRR bit, while the latter method relies on level-sensing circuitry. The trigger method employed varies between different types of computer system. It should not be changed by the user. ISA and XT bus machines program the PIC to respond to

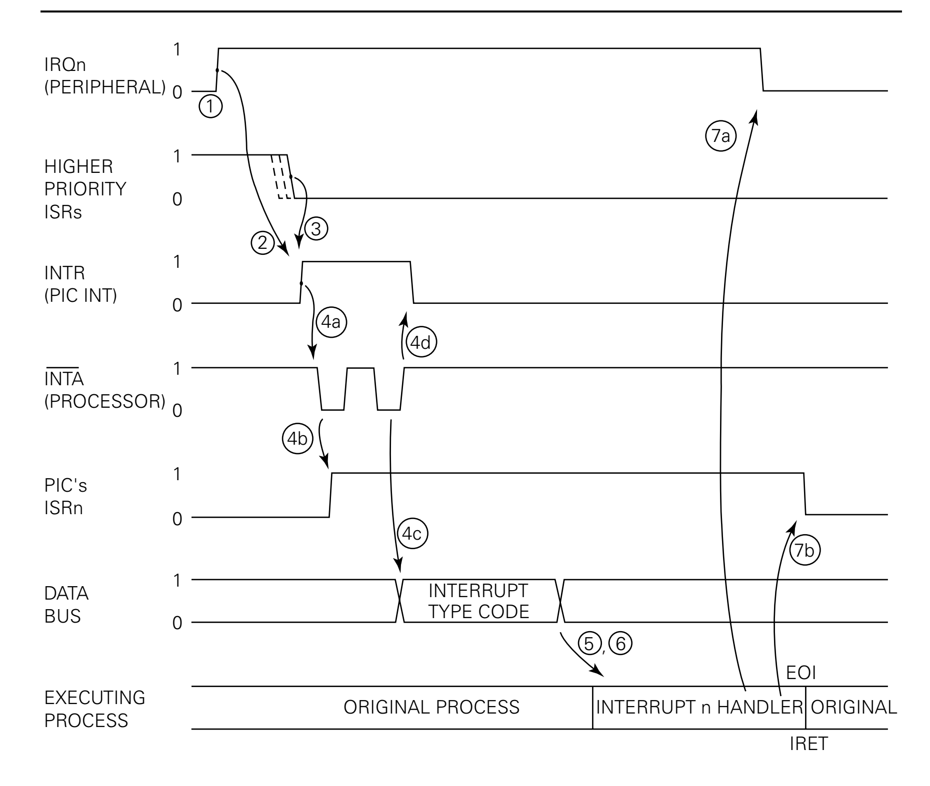


Figure 5.3 *The interrupt sequence*

edge-triggered interrupts while MCA machines (i.e. most PS/2s) use level-triggered interrupts. EISA machines default to edge triggering for compatibility with AT systems, but may also be programmed for level-triggered interrupts.

In an edge-triggered system, an interrupt is generated only when the IRQ line first undergoes a low-to-high transition. The line may remain high without further interrupts being triggered. However, if the IRQ stays high in a level-triggered system, a second interrupt will be generated as soon as the software issues an EOI command to acknowledge the first interrupt. It is, therefore, essential to deactivate the IRQ line before issuing an EOI to a level-triggered PIC.

One consequence of level-triggered interrupts is that they facilitate sharing of IRQ lines between different devices. MCA machines incorporate hardware that allows more than one peripheral device to drive the same IRQ line. The IRQ remains asserted as long as one or more peripherals are requesting service. To accommodate this mode of operation, each peripheral must provide a software-readable flag to indicate when it requires service. The interrupt

handler routines associated with each device on the shared IRQ line are installed in a chain-like structure. The first handler to gain control when an interrupt occurs should check whether its own associated device requires attention. If it does not, the handler must immediately call the previous interrupt handler in the chain (i.e. the one associated with the next device attached to the shared IRQ). This process repeats until all devices that require attention have been serviced.

Although this method provides additional scope for system expansion, it does increase the overall time taken to respond to interrupts. In some DA&C applications this additional delay might unacceptably compromise the real-time performance of the system. In general, it is wise to avoid using shared interrupts for any subsystem that requires a fast interrupt response. Interrupt response times and latencies are discussed in the section *Interrupt response times* at the end of this chapter.

Cascaded PICs on ISA and MCA machines

In order to expand their interrupt processing capability from 8 to 15 IRQ lines, ISA- and PCI-based PCs (i.e. AT compatibles) and MCA machines (i.e. IBM PS/2s) are equipped with two 8259A PICs, connected together in a cascaded configuration. This requires the BIOS's Power-On Self Test (POST) routines to initialize the PICs in a slightly different manner so that they will operate as a master and slave. The primary (master) PIC is used in much the same way as on PC and XT machines and is mapped to the same I/O addresses (ports 20h and 21h). The secondary (slave) PIC appears at ports A0h and A1h. The eight interrupt request lines provided by the additional PIC are referred to as IRQ8–IRQ15. The slave's INT output line is fed to the IRQ2 input of the master PIC. In this way any interrupt requests occurring on IRQ8–IRQ15 result in an interrupt being signalled on the master PIC's level 2 input. This has obvious consequences for the interrupt priority scheme described previously. Figure 5.4 illustrates how the two PICs are connected.

When the slave receives an interrupt request, it prioritizes it in the same way as previously described and asserts its INT line. This is detected by the master PIC on its IRQ2 line. The master then prioritizes this interrupt request and asserts the processor's INTR pin. When the processor responds with two INTA pulses, the master PIC effectively passes control to the slave by means of the CAS0 to CAS2 lines. These enable the slave and cause it (rather than the master PIC) to place an Interrupt Type Code (usually in the range 70h to 77h) on the data bus during the second INTA cycle.

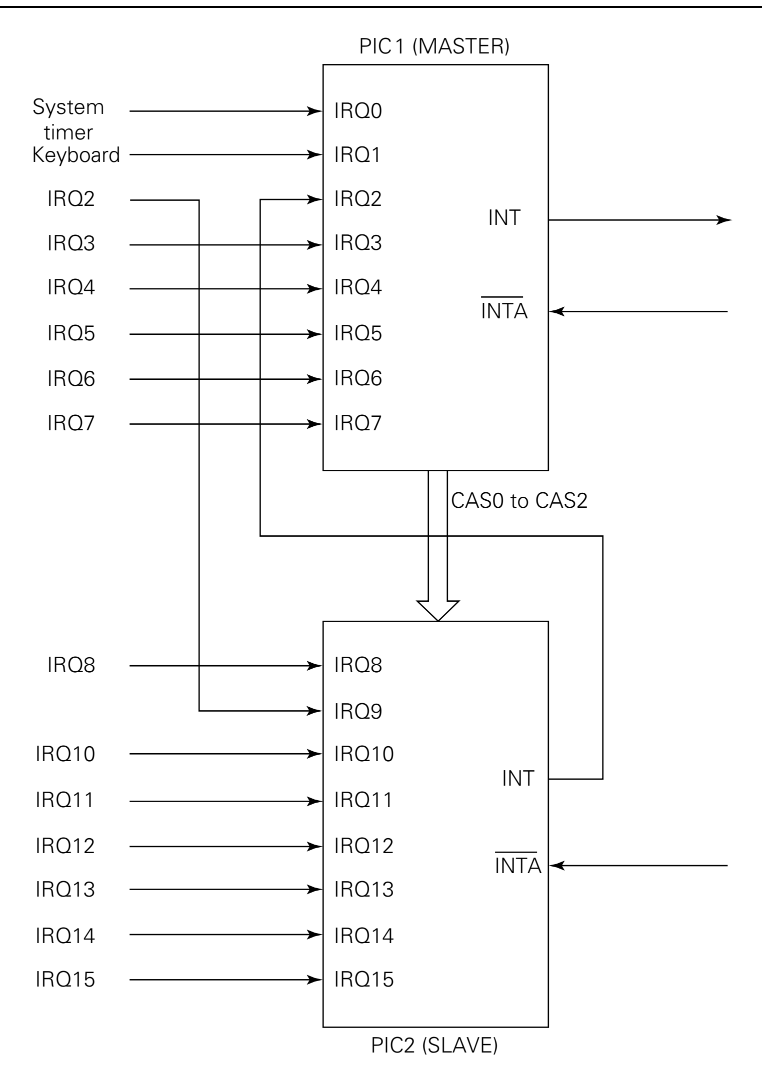


Figure 5.4 *Cascaded master and slave PICs on the IBM AT and PS/2*

If an interrupt originates from the slave PIC the interrupt handler routine must issue EOI commands to both PICs before exiting: the slave should be acknowledged first and the master second. Note that further interrupt requests made via the slave PIC will not be recognized by the master until after the master has received an EOI command.

Because the master PIC's level 2 input is connected to the INT output from the slave PIC, the IRQ2 line is no longer available

to accept interrupt requests. The modern AT-compatible PCs are designed to maintain compatibility with the IBM PC and XT (which were able to make use of IRQ2) by connecting the IRQ2 line on the expansion bus to the slave's IRQ9 input. IRQ9 is mapped to the interrupt 71h vector. The BIOS incorporates an interrupt 71h handler which simply makes a software call to the interrupt 0Ah (IRQ2) handler. In this way, if an adaptor card issues an interrupt request on the IRQ2 expansion bus line, the correct interrupt handler is still invoked (although the interrupt request is routed through IRQ9 and the slave PIC instead of going directly to the master). This allows software and hardware designed for earlier systems to work without modification on AT-compatible PCs.

It is interesting to note that it is possible to expand the PC's interrupt system by interfacing additional PICs. Because some of the interface lines required for cascading the additional PICs are not available on the expansion bus, full cascading is not possible. Software interrupt handlers must, in this case, poll the various PICs in order to determine which device requested service. This technique is described in detail by Eggebrecht (1990).

Remapping interrupts

During the second INTA cycle, the PIC passes an 8-bit Interrupt Type Code to the processor. This code is actually the ordinal index of the interrupt vector which is to be used to transfer control to the interrupt handler. Bits 0 to 2 of the Interrupt Type Code contain a binary-coded representation of the number (in the range 0 to 7) of the IRQ line which generated the interrupt. The 5 highest order bits determine which of the available 256 interrupt vectors are mapped to the IRQ lines. These bits are programmed into the PIC during initialization (i.e. usually by the BIOS's POST routines). This allows the system initialization code to map the block of eight interrupt lines associated with each PIC to a specific region of the IVT. For the master PIC present on all PC-compatible machines, the high order 5 bits of the Interrupt Type Code are such that IRQ0–IRQ7 are mapped to interrupts 08h–0Fh. The value programmed into the secondary PIC (on ISA, PCI, EISA and MCA machines only) routes IRQ9–IRQ15 to interrupts 70h–77h. The interrupts may be remapped simply by reinitializing the PIC(s) with a suitable value for the 5 high order bits of the Interrupt Type Code. Remapping hardware interrupts in this way might introduce incompatibilities with software which expects the IRQs to invoke the usual interrupts. If you do remap the interrupts be sure to account for any such incompatibilities and remember to redirect the new interrupts to the appropriate interrupt handlers.

Programming the PIC and reading its registers

The 8259A PIC is a very flexible device and may be programmed to operate in a variety of modes. Some of these modes are not compatible with the PC's architecture, or even with the 80x86 family of processors, so you will need to exercise great care if you wish to reprogram this device.

As explained previously, the system BIOS's POST routines configure the PIC to a standard operating mode, and there is usually no need for the programmer to subsequently reprogram the device. Indeed to do so may affect the ability of the PIC to function correctly in conjunction with BIOS and other system components. Nevertheless, there are rare instances when it is necessary to change the PIC's operating mode and so the relevant commands are discussed briefly below. We will, however, discuss only those modes and commands that are useful on the PC. You should refer to Intel's 8259A Programmable Interrupt Controller Data Sheet for additional programming details.

Mode selection and other commands may be issued to the PIC either as an initialization sequence of 2 to 4 bytes – known as Initialization Command Words (ICWs) – or subsequently as individual Operational Command Words (OCWs). The PIC has two 8-bit ports, each of which accepts certain command words: these are detailed below. We will refer to these ports as port 0 and port 1. On the master PIC, ports 0 and 1 are mapped to I/O addresses 20h and 21h respectively. The slave PIC present on ISA, PCI, EISA and MCA systems uses ports A0h and A1h.

Initialization command sequence

An application program may reinitialize the PIC if it wishes to modify certain modes of operation. Initialization involves the software writing from two to four Initialization Command Words to ports 0 and 1. The first ICW, known as ICW1, is written to port 0. Bit 4 of ICW1 is always set and this allows the PIC to distinguish it from Operational Command Words which all have bit 4 reset (i.e. 0). The values of bits 0 and 1 of ICW1 determine whether the third and fourth ICWs are needed. Note that the format of ICW3, if needed, depends upon whether the PIC has been configured as a master or as a slave.

It is not usually necessary to reinitialize the PIC because the BIOS POST routines will normally have set the device to the correct operating mode. Consequently initialization will not be discussed in detail here and the bit assignments listed in Tables 5.2 to 5.6 will be presented without further comment. If you need additional

Table 5.2 *ICW1, for output to port 0*

Bit	Name	Description
0	IC4	1 = Use ICW4. If this bit is 0, ICW4 would not be required and the functions controlled by ICW4 would be treated as though all ICW4 bits were 0.
1	SNGL	1 = No cascade (used on PC and XT). ICW3 is omitted. 0 = Cascade mode (used on AT and PS/2). ICW3 is required.
2	ADI	Always 0. Unused on PC and compatibles.
3	LTIM	1 = Level-triggered IRQs (MCA machines). 0 = Edge-triggered IRQs (PC, XT, AT systems).
4		Always 1. Identifies the command as being ICW1.
5–7	A5–A7	Always 0. Unused on PC and compatibles.

Table 5.3 *ICW2, for output to port 1*

Bit	Name	Description
0–2	A8–A10	Always 0. Unused on PC and compatibles.
3–7	T3–T7	High order 5 bits of the Interrupt Type Code that is transferred to the processor during the second INTA cycle. Master PIC uses 00001b and slave PIC uses 01110b.

Table 5.4 *ICW3, for output to port 1 of the master PIC*

Bit	Name	Description
0–7	S0–S7	Each bit represents an interrupt level used to cascade to a slave PIC. Each bit set to 1 indicates that a slave PIC is attached to the corresponding IRQ level. On the AT, IRQ2 is used for cascading the slave PIC so ICW3 is 00000100b.

Table 5.5 *ICW3, for output to port 1 of the slave PIC*

Bit	Name	Description
0–2	ID0–ID2	ID code of slave device (same as master's IRQ level to which the slave is attached): 010b on AT.
3–7		Always 0. Unused on PC and compatibles.

Table 5.6 *ICW4, for output to port 1*

Bit	*Name*	*Description*
0	μPM	Always 1. Indicates 80x86 compatibility mode.
1	AEOI	Always 0. Indicates no automatic EOI.
2	M/S	Always 0.
3	BUF	Always 1 on PC and XT. Indicates buffered mode. Always 0 on AT. Indicates non-buffered mode.
4	SFNM	Always 0. Indicates not special fully nested mode.
5–7		Always 0. Unused on PC and compatibles.

Table 5.7 *Summary of useful 8259A PIC operational commands*

	Port		
Command code	*Master PIC*	*Slave PIC*	*Description*
0Ah	20h	A0h	Map IRR to port 20h/A0h for reading.
0Bh	20h	A0h	Map ISR to port 20h/A0h for reading.
20h	20h	A0h	Non-specific end of interrupt (EOI).
C0h–C7h	20h	A0h	Set priority.
Mask	21h	A1h	Set interrupt mask (load IMR).

information, you should consult the Intel 8259A Programmable Interrupt Controller Data Sheet.

Operational commands

After the PIC has been initialized by the BIOS POST routines, various operational commands may be issued to the PIC in order to perform actions such as reading the ISR or acknowledging an interrupt. We have already introduced some of the operational commands: accessing the IMR and issuing a non-specific end-of-interrupt (EOI), for example. A number of other useful commands are available to the programmer. These allow the software to read the PICs' status registers (i.e. the IRR and ISR) and to select various operating modes. A selection of Operational Commands are listed in Table 5.7. Unlike the Initialization Commands, the Operational Command Words do not need to be issued in sequence. Note that any interruptible command sequence (e.g. reading the IRR) should be carried out with processor interrupts disabled.

Map IRR to Port 0 command (write 0Ah to port 0 (20h/A0h))

This command maps the IRR to port 0 so that subsequent reads from I/O port 20h (or A0h for the slave PIC) will return the contents of

the IRR. Each of the eight IRQ inputs is represented by 1 bit of the IRR: bit 0 indicates whether an IRQ0 request is pending; bit 1 indicates whether an IRQ1 request is pending and so on. All pending interrupt requests are denoted by a 1 bit. It is sometimes useful to read the IRR in order for an interrupt handler to check whether any lower priority interrupts are pending. Other software routines can also use this facility to determine whether an interrupt request has occurred while external hardware interrupts may have been masked.

Map ISR to Port 0 command (write 0Bh to port 0 (20h/A0h))

This command maps the ISR to port 0 to that subsequent reads from I/O port 20h (or A0h for the slave PIC) will return the contents of the ISR. The ISR contains 1 bit for each possible IRQ level in much the same way as the IRR. However, a high ISR bit indicates that the corresponding interrupt level is currently being *serviced* (i.e. the interrupt has been invoked, but the handler has not yet issued an EOI). All interrupts which are currently in service will be represented by high ISR bits. Only one bit of the ISR will usually be set during execution of an interrupt handler, but if one or more higher priority requests have interrupted a lower priority handler before the latter has issued an EOI (and thus cleared its associated ISR bit), more than one ISR bit will be set. Reading the ISR also provides a means for a shared interrupt handler (e.g. one written to handle input from two or more serial ports) to determine which device issued the interrupt.

Non-specific End-of-Interrupt command (write 20h to port 0 (20h/A0h))

The non-specific EOI command should be issued by each interrupt handler before returning control to the interrupted process. This command clears the ISR bit corresponding to the highest priority interrupt currently in service. This will normally be the interrupt which issued the EOI command. By clearing the ISR bit, the command allows further interrupts of equal or lower priority to occur. On dual-PIC systems (e.g. ISA, PCI or MCA), any interrupt handlers which are invoked via the slave PIC (i.e. via IRQ8–IRQ15) must issue EOI commands to both PICs. The slave PIC should be acknowledged first and then the master.

Set Priority command (write C0h-C7h to port 0 (20h/A0h))

This set of commands allows different priorities to be assigned to each IRQ input. Normally, the PIC is programmed to allocate IRQ0 requests the highest priority and IRQ7 the lowest. Table 5.8

Table 5.8 *Interrupt priorities defined by the set priority command*

	IRQ priority order							
Priority	*C0h*	*C1h*	*C2h*	*C3h*	*C4h*	*C5h*	*C6h*	*C7h*
1 (highest)	1	2	3	4	5	6	7	0
2	2	3	4	5	6	7	0	1
3	3	4	5	6	7	0	1	2
4	4	5	6	7	0	1	2	3
5	5	6	7	0	1	2	3	4
6	6	7	0	1	2	3	4	5
7	7	0	1	2	3	4	5	6
8 (lowest)	0	1	2	3	4	5	6	7

illustrates the priorities assigned to each IRQ input by the Set Priority commands. Note that, in the case of the slave 8259A PIC, the interrupt request levels listed as 0–7 actually refer to IRQ8–IRQ15.

Suppose that it is necessary to incorporate a section of time-critical code within a DA&C program. It may be desirable in some situations to install the code within a high priority interrupt handler. This prevents other external hardware interrupts from taking precedence and thereby delaying execution of the code. The hardware which is to generate the interrupt requests might, for example, be connected to IRQ7. In the case, the C6h command would be issued. This would allocate the highest priority to the new IRQ7 process: higher than even the system clock interrupt on IRQ0. You should exercise great care when reassigning interrupt priorities and should be aware of all possible consequences of doing so. You should also confine any high priority processes to as short a time span as possible in order to avoid adversely affecting other interrupt-based subsystems.

Define Interrupt Mask command (write mask byte to port 1 (21h/A1h))

It is possible to modify the Interrupt Mask Register (IMR) by writing to this port. The IMR may also be read by reading from port 1. Each bit masks or unmasks the corresponding interrupt level. Bit 0 is associated with IRQ0, bit 1 with IRQ1 etc. Each low bit in the IMR enables the corresponding IRQ level and a high bit disables the IRQ.

The Non-Maskable Interrupt

The processor's Non-Maskable Interrupt (NMI) facility provides a means for the various PC subsystems to notify the processor when

some critical event, such as a hardware failure, has been detected. On ISA and XT-bus machines, there are three possible sources of NMIs: RAM parity failure, I/O channel error or a numeric coprocessor error. On MCA systems, channel 3 of the system timer (i.e. the watchdog timer) can also initiate an NMI. There are a number of additional sources of NMIs on EISA machines.

One important difference between NMIs and external hardware (INTR) interrupts is that the processor does not attempt to retrieve an Interrupt Type Code from the data bus. Instead, it always uses interrupt type 2 to service the NMI. This is a fixed feature of the processor and cannot be changed by the programmer.

NMI handler routines are normally implemented by the system BIOS. In situations such as a memory parity error, the BIOS's NMI handler will usually display a message to indicate the nature of the fault. In such cases there is generally no way to recover reliably from the problem and so the BIOS closes down the system.

NMIs have the highest priority of all hardware interrupts and this guarantees a more or less immediate response to a pending error condition. The only conditions that can delay execution of an NMI are:

- The NMI has been disabled by software (e.g. by code that reads the CMOS RAM or Real Time Clock).
- The processor is responding to a higher priority interrupt (such as an exception).
- The processor has begun execution of an instruction that changes the SS (stack segment) register. In this case the NMI will not be recognized until after the following instruction has been executed.

Enabling and disabling the NMI

As its name suggests, and unlike external interrupts on the INTR line, the NMI cannot be masked (disabled) within the processor itself. However, the AT and compatible machines incorporate circuitry for gating off the NMI signal before it reaches the processor. The BIOS POST routines ensure that the NMI is enabled during start-up, so that any subsequent memory or I/O errors will generate an NMI. An application program may disable the NMI by setting bit 7 of I/O port 70h to 1. The NMI may be re-enabled by clearing the same bit. Port 70h is also used to access the AT's Real Time Clock and CMOS RAM. The NMI should normally be disabled in this way whenever you attempt to read from, or write to, the CMOS RAM. It is generally inadvisable to disable the NMI for an appreciable length of time.

Signalling a system failure

In most DA&C applications it is unnecessary to install your own NMI handling routines. If a RAM parity or other critical error occurs, there is little that the programmer can do to recover. However, there are situations where you might wish to inform an external device of the fault by, for example, closing a relay or otherwise asserting a digital I/O line. This might be facilitated by intercepting the NMI, but this technique will not normally be foolproof. There are likely to be many other possible (and more probable) failure modes in a typical data-acquisition system: obvious examples are loss of power or a software crash due a coding error. If it is necessary to inform external equipment of a general system failure, it will usually be more reliable to make use of a watchdog timer as described in Chapter 3. If you need to write your own NMI handlers you may wish to consult the text by van Gilluwe (1994) which provides further information on this topic.

It should be noted at this point that you should not rely on the PC, its software or peripheral devices to control or monitor a potentially hazardous system. Reliable as most modern PCs are, they are very complex machines and, as a general rule, the more complex a system is, the more scope there is for it to fail! Any PC-based DA&C system should always be supplemented by whatever fail-safe mechanisms might be necessary to ensure total safety. This point may (indeed, should) be obvious to the reader, but it is of such importance that it cannot be overemphasized.

5.3 Software interrupts and processor exceptions

Software interrupts and processor exceptions are both generated by events which occur within the confines of the processor itself. They arise as a result of the processor executing a specific instruction or sequence of instructions. Software interrupts may be initiated by special interrupt instructions placed in the program. They are generally used to provide a means of communicating with other software processes such as DOS or the PC's BIOS. Processor exceptions, on the other hand, generally arise from some form of error condition, such as an attempt to divide a number by zero.

Software interrupts

Software interrupts are used on the PC as a way of implementing address-independent interprocess software calls. Many PC programs

use the software interrupt mechanism for accessing the BIOS and operating-system services.

The interrupt sequence

Because software interrupts are generated by interrupt instructions placed within a program sequence they always operate synchronously with the processor. Consequently the precautions outlined previously in regard to accessing global data structures and other shared resources do not apply. In other respects, however, the operation of the two types of interrupt are very similar. On encountering a software interrupt instruction, the processor pushes the Flags register, clears the Interrupt Flag (IF) and the Trap Flag (TF) and then pushes the CS and IP registers onto the stack. During this process the processor also retrieves the address (CS:IP) of the interrupt handler from the IVT and then transfers control to the handler. After all necessary processing has been completed, the interrupt handler should return control to the calling process by issuing an IRET instruction. Because the interrupt was generated within the processor, there is, of course, no need to acknowledge the PIC with an EOI command.

The Interrupt Type Code (i.e. the index into the IVT) is usually obtained from the interrupt (INT) instruction itself. A few instructions (such as the INTO and BOUND instructions or the Breakpoint opcode) will only generate an interrupt under specific conditions. The Interrupt Type Code used in these cases is not received from the instruction sequence, but is instead generated by the processor. We will not discuss these instructions here. See Hummel (1992) or your assembly language programming manuals for further information on these interrupts.

When a software interrupt occurs, the processor always clears the Interrupt Flag immediately after pushing the original Flags register onto the stack. This means that all maskable (i.e. external hardware) interrupts will be disabled until the interrupt handler either issues an STI instruction or returns with an IRET (which restores the original contents of the Flags register). Unless there is a good reason to do otherwise, it is sensible for a software interrupt handler to unmask the external hardware interrupts (i.e. issue an STI instruction) as soon as it gains control. Software interrupts have a higher priority than either of the hardware (INTR or NMI) interrupts. Note that software interrupts are not maskable and so are not affected by the state of the Interrupt Flag.

The interrupt sequence in protected mode is similar in many respects, although there are some important differences. See Hummel (1992) for more on protected-mode interrupts.

Issuing a software interrupt in assembly language

A software interrupt may be invoked from an assembly language program by means of the 2-byte INT instruction. The first byte is always the CDh opcode and the second byte may be any number from 0 to 255: this is actually the Interrupt Type Code which the processor uses to retrieve the associated interrupt vector. The INT instruction is capable of invoking any available interrupt, even one reserved for a processor exception or hardware interrupt. The following real-mode code fragment illustrates how interrupt 21h (the DOS Function interrupt) may be called from an assembly language program. This particular example calls the Get DOS Version function, as denoted by the value of 30h placed into the AH register, and then checks to see whether it is version 3.0 or later.

```
mov  ah,30h             ;Get DOS Version function number
int  21h                ;Call DOS using software interrupt
cmp  al,3               ;Is it version 3.0 or later ?
jge  DOSVersionOK       ; - Yes, proceed
jmp  DOSVersionError    ; - No,  jump to error routine.
```

The details of calls to other functions (i.e. the register usage) will differ, but the same interrupt call mechanism applies.

Note that the actual value of the Interrupt Type Code (in this case 21h) is coded into the instruction sequence. It is not possible to code an interrupt call using a variable Interrupt Type Code. If you wish to do this you will need to build a table of `int` instructions and then use the Interrupt Type Code as an index for jumping into the table. A more efficient, but in some ways a less satisfactory, alternative is to use self-modifying code – i.e. software that writes the Interrupt Type Code directly into the instruction sequence in memory prior to executing the `int` instruction. It is often inadvisable to use this technique, however. One has to account for the operation of caches and prefetch queues within the processor and circumvent problems with writing to the code segment in protected mode. Self-modifying code can also be difficult to debug and cannot be run from ROM – e.g. in embedded applications.

For further information on the prefetch queue and protected mode programming refer to Hummel (1992). A discussion of interrupts under Microsoft Windows can be found in the text by Solomon (1998).

Issuing a software interrupt from a high level language

Many compiled high level languages such as C and Pascal include functions or procedures for issuing software interrupts. A jump

table, or self-modifying code similar to that described above allows the function to receive the Interrupt Type Code as a variable parameter. Although not defined by the ANSI C standard, compilers such as Borland C provide the `int86()` and `int86x()` functions for invoking software interrupts (refer to your programming language technical manual for further information on these functions). Other languages provide similar functions: Borland Pascal, for example, has a procedure known as `Intr()`. In all cases these functions or procedures allow the calling process to pass data to the interrupt handler via the processor's registers and to receive any results back in the same way. The registers are encoded in a data structure such as a union in C or a variant record in Pascal.

The following code fragment illustrates how the C language's `int86()` function may be used to call a BIOS service. In this example, we invoke the service which moves the cursor to position X,Y on the display screen.

```
void SetCursorPos(unsigned char X, unsigned char Y)
/* Changes the text screen cursor position on page 0.*/
{
union REGS In, Out;

In.h.ah = 0x02;
In.h.bh = 0x00;
In.h.dl = X - 1;
In.h.dh = Y - 1;
int86(0x10, &In, &Out);
}
```

The `h` qualifier in the `In.h.dl = X - 1` line, for example, provides access to byte-sized registers. To access a word register, such as DX, it would be necessary to use `In.x.dx = ...` etc. Hexadecimal constants are denoted by the `0x` prefix in C, so in this example the `int86(0x10...)` instruction actually calls interrupt 10h: the BIOS video services. Note that the *addresses* of the `In` and `Out` register structures are passed to the `int86()` function as denoted by the `&` prefixes.

A number of other interrupt functions and procedures are available for making calls direct to DOS using interrupt 21h. Borland C provides the `intdos()` and `intdosx()` functions for this purpose. Similar functions are available in other high level languages.

Processor exceptions

Processor exceptions are generated internally by the processor as a result of executing a specific sequence of instructions. They are generally used to signal some form of error condition. As they

are not generated independently of the processor, exceptions are always synchronous. Like software interrupts, processor exceptions cannot be masked. They have the highest priority of all types of interrupt: higher even than the NMI. Most types of exception are only generated in protected mode or V86 mode. A full discussion of processor modes and exceptions is beyond the scope of this book. Interested readers are referred to the text by Hummel (1992) which provides a very detailed account of this topic.

5.4 Interrupt priorities

The priorities which the processor and PIC assign to the various types of interrupt have already been mentioned. A high priority interrupt request will, if it occurs simultaneously with one of a lower priority, be recognized first. Lower priority interrupts are generally inhibited until the interrupt handler acknowledges the source of the interrupt, issues an EOI command to the PIC and, if necessary, sets the processor's Interrupt Flag. Table 5.9 illustrates the default prioritization applied by the 8259A PIC(s) to the various external hardware interrupts.

Note that although this prioritization is implemented by the PC's hardware, it is possible for software to modify the effective priorities

Table 5.9 *Normal external hardware interrupt priorities of the 8259A PIC*

Priority	*PC and XT*	*AT, PS/2 and EISA*
1 (highest)	IRQ0: System timer	IRQ0: System timer
2	IRQ1: Keyboard	IRQ1: Keyboard
3	IRQ2: LPT2/Reserved*	IRQ8: Real-time clock
4	IRQ3: COM2*	IRQ9 (labelled IRQ2 on bus): Reserved*
5	IRQ4: COM1*	IRQ10: Spare*
6	IRQ5: Hard disk controller*	IRQ11: Spare*
7	IRQ6: Diskette controller*	IRQ12: Spare (AT); Pointing device (PS/2)*
8	IRQ7: LPT1*	IRQ13: Coprocessor*
9		IRQ14: Hard disk controller*
10		IRQ15: Spare*
11		IRQ3: COM2*
12		IRQ4: COM1*
13		IRQ5: LPT2 (AT): Reserved (PS/2)*
14		IRQ6: Diskette controller*
15 (lowest)		IRQ7: LPT1*

*Available on expansion bus.

of the interrupts by reprogramming the PIC(s) as described in *Programming the PIC and reading its registers* earlier in this chapter.

The processor itself must prioritize all interrupts that it receives – i.e. hardware interrupts occurring on the INTR line together with the NMI, processor exceptions, traps and software interrupts. The processor's prioritization scheme varies with the type of processor and with the state of its flags, and in some cases also depends upon which combination of interrupt requests are pending. In general though, certain processor faults (e.g. divide-by-zero errors) and traps (e.g. debug trap) have the highest priority, and external hardware interrupts have the lowest (although the 80486 and later processors assign even lower priorities to some faults and exceptions). Unmaskable interrupts, including the NMI, software interrupts and processor exceptions have intermediate priorities. The details of the various processors' prioritization schemes are beyond the scope of this book. Interested readers are referred to Hummel (1992) for further information.

The point of this discussion is that the NMI, some types of trap and software interrupts can take precedence over external hardware interrupts. This has obvious implications for developers of real-time systems where the presence of higher priority interrupts might adversely affect interrupt latencies.

5.5 Writing interrupt handlers

Interrupt handlers have a multitude of applications within DA&C software. They can, for example, be used to enable the processor to read an ADC or the serial port whenever new data becomes available. They are also commonly used for timekeeping and pacing. Periodic interrupts from the system timer or from an external device allow the software to perform actions at regular intervals. These actions might include tasks such as checking the status of a limit switch or relay (via an I/O port) or controlling an actuator. Various PC subsystems can be manipulated by hooking interrupts. For example, it is possible to detect or filter out specific key combinations (such as Ctrl-Alt-Del) by intercepting the keyboard interrupt.

Finally, and perhaps most importantly, the interrupt system allows the programmer to trap specific error conditions (e.g. a divide by zero) and events such as a Ctrl-C or Ctrl-Break interrupt. The application software can install routines to handle the error and to provide a suitable recovery mechanism. This consideration is generally of most importance to assembly language programmers since most high level languages (HLLs) incorporate mechanisms

for automatically trapping these interrupts. Nevertheless, all users of HLLs should be familiar with the error trapping facilities of their compiler. This topic is covered adequately in many books on DOS programming (e.g. Duncan (1988) and Dettmann and Johnson (1992)) and so will not be discussed here.

The following subsections describe how interrupt handlers can be installed in a real-mode data-acquisition program. They also illustrate how the functionality of existing interrupt handlers may be preserved by adding new handlers in a chain-like structure. Similar principles will apply to interrupts in protected mode, but you should be aware that the structure of the interrupt handler may be governed by the operating system in use. Indeed the operating system may even hide the mechanics of the interrupt process from the application. Windows NT, for example, allows only privileged operating-system code or device drivers to directly handle interrupts, although there are callback facilities that allow less privileged user-mode code to be invoked indirectly as a result of an interrupt.

Additional information on using the PC's interrupt system in real-mode is provided in the texts by Swan (1989) and Holzner and Norton (1991). Solomon (1998) describes interrupt processing under Windows NT in some detail.

Installing an interrupt handler

In order to install an interrupt handler, the corresponding interrupt vector must be modified so that it points to the new routine. Before doing this, however, the original value of the interrupt vector should be recorded so that it can be restored before the program terminates. A record of the original interrupt vector is also essential in cases where control must be passed to the old interrupt handler. There are two ways in which the individual interrupt vectors may be modified: via operating system functions or by directly accessing the IVT in low memory. For reasons of simplicity and portability, the former method is normally to be preferred. In fact, a number of high level languages provide library functions which are based on these services. Borland's implementations of C provide the `getvect()` and `setvect()` functions for reading and modifying interrupt vectors.

However, there are circumstances, in a real-mode program, where it is preferable to read from, or write to, the IVT directly. This is often perfectly acceptable provided that there is no possibility of an interrupt occurring while the IVT is being accessed. It is usually safest to disable all hardware interrupts during IVT accesses. The IVT is 1024 bytes long and, in real mode, is located at the very bottom of the PC's memory (i.e. at 0000:0000h). Each vector occupies 4 bytes

and so the offset of the vector with type code n is at $4n$. Vector 0 is at offset 0000h, vector 1 is at offset 0004h, vector 2 is at offset 0008h and so on.

Masking and unmasking the interrupt

If you are installing a handler for an external hardware interrupt it may be necessary to unmask the associated IRQ by modifying the contents of the PIC's IMR. This action will, of course, be required only if the interrupt was previously unused. If the new handler is intended to replace, or link into, an existing interrupt handler, the IRQ level will already be unmasked and it will not be necessary to modify the IMR.

Each bit of the IMR corresponds to one IRQ line: bit 0 is associated with the level 0 interrupt, bit 1 with the level 1 interrupt and so on. Each zero IMR bit causes the corresponding IRQ level to be unmasked (enabled). Note that you can read the IMR from I/O port 21h (or A1h in the case of the secondary PIC) in order to determine which interrupts are presently enabled. Only the bit corresponding to the desired interrupt should be modified. Because many of the remaining IRQ levels are used by other subsystems, masking or unmasking these interrupts may have undesired effects. It is wise to take the precaution of disabling interrupts (with a CLI instruction) while accessing the PIC's IMR. The example in Listing 5.2 illustrates how to modify the IMR.

The structure of the interrupt handler

The basic structure of software and hardware-interrupt handler routines is quite simple. In both cases, the handler must first save the contents of all of the processor's registers so that they can be restored before exiting. If the registers are not preserved in this way, it is likely that the interrupt handler will corrupt data belonging to the interrupted process. The usual technique is to save the registers on to the stack as shown in Listing 5.1. Obviously, only those registers which are actually modified by the interrupt handler need to be saved and restored.

After saving the registers, the handler may service the interrupt and carry out whatever processing is necessary. In the case of a hardware interrupt handler, the code should usually acknowledge the device which caused the interrupt so that it deactivates its interrupt request line.

Listing 5.1 *Basic interrupt handler shell*

```
   PROC    IntHandler FAR
   ;
   ; General purpose interrupt handler shell.
   ;
                push   ax                     ; Save registers on stack
                push   bx                     ;
                push   cx                     ;
                push   dx                     ;
                push   di                     ;
                push   si                     ;
                push   bp                     ;
                push   es                     ;
                push   ds                     ;


                ; Perform interrupt processing here


                pop    ds                     ; Restore regs. from stack
                pop    es                     ;
                pop    bp                     ;
                pop    si                     ;
                pop    di                     ;
                pop    dx                     ;
                pop    cx                     ;
                pop    bx                     ;
                pop    ax                     ;

                iret                          ; Return from interrupt

ENDP    IntHandler
```

Returning from the interrupt and restoring the interrupt flag

When the interrupt is invoked, the processor pushes the Flags register and the CS and IP registers on to the stack before transferring control to the interrupt handler. The handler can easily read the return address by accessing the appropriate location in the stack segment. This technique is useful for handling some processor exceptions and for creating profiling routines. Note that if you are writing interrupt handlers in a language such as C or Pascal using high level interrupt-type functions or procedures, the compiler will automatically save and restore the registers for you. The order in which they are pushed onto the stack may, however, differ from that shown in Listing 5.1.

When a software or hardware interrupt handler first gains control, the processor's Interrupt Flag (IF) will be clear so no further external hardware interrupts will be recognized until after the handler terminates with the IRET instruction. Depending upon the nature of the

application, you may wish to unmask the interrupts by issuing an STI instruction at an earlier point within the handler.

When external hardware interrupts are unmasked by means of the STI instruction or by restoration of the Flags register during an IRET, any pending INTR requests will remain unrecognized until after the instruction which follows the STI or IRET! This facility allows the programmer to prevent multiple interrupt handlers from being called in a nested fashion. It therefore helps to eliminate excessive stack usage, by keeping further interrupts disabled until after the final IRET instruction has been executed.

When writing a *software* interrupt handler, you may need to return status information or other data in the Flags register. In this case you should not use an IRET because this instruction would overwrite the new Flags status with the original contents of the Flags register! The handler should, instead, unmask interrupts and exit with an RETF 2 instruction which will leave the new contents of the Flags register intact. Some system interrupts, such as DOS interrupt 21h, use this technique to return information in the Flags register. Remember, however, that this technique only applies to software interrupt handlers. You should, of course, *always* use IRET to return from any interrupt handler that is entered asynchronously (i.e. a hardware interrupt handler).

Hardware interrupt handlers

Unmasking the processor's Interrupt Flag will allow only interrupts of a higher priority than the one currently executing to be recognized. To allow lower priority interrupts to execute it is necessary to issue a non-specific EOI command to each of the PICs involved in the interrupt request:

```
; Send EOI commands to PICs
mov    al,20h                   ; Non-specific EOI command
out    0A0h,al                  ; Send EOI to slave PIC
out    20h,al                   ; Send EOI to master PIC
```

If the interrupt request is not routed through the slave PIC (i.e. on XT-bus systems or on ISA systems if the interrupt is on IRQ0–IRQ7), the `out 0A0h,al` line is not required and should be omitted.

The EOI command clears the ISR bit that corresponds to the current interrupt, which allows lower priority interrupt requests to be serviced. Even if you are content with keeping low priority interrupts disabled, the EOI command should always be issued at some point within the interrupt handler. It is possible to determine whether other interrupt requests are pending or currently in service

by reading the PIC's IRR and ISR as described in the section *Programming the PIC and reading its registers* earlier in this chapter.

Listing 5.2 illustrates how a handler routine may be implemented in C for an external hardware interrupt. This example installs a handler for interrupt 0Dh (IRQ5), but can easily be adapted for other interrupts.

The `interrupt` keyword available in Borland and Microsoft implementations of C informs the compiler that the associated function is an interrupt handler. This causes the compiler to generate special entry and exit code for the function which preserves the contents of the processor's registers and terminates the routine using an `IRET` instruction. The entry and exit code is similar, although not identical, to that shown in Listing 5.1. When an interrupt function is called, the DS register is initialized to point to the program's data segment (in medium memory models), and this allows the interrupt handler

Listing 5.2 *Installing an interrupt handler for interrupt 0Dh (IRQ5)*

```
#include <dos.h>

unsigned char OrigIMR;                    /* Original PIC int mask register */
void interrupt (*OrigIntDVector)(void);   /* Storage for orig int 0Dh vector */
        :
        :
/* Function Prototypes */
void InstallIntDHandler(void);
void RestoreIntDHandler(void);
        :
        :
void interrupt IntDHandler()
{
/* Do any required processing here */
outportb(0x20,0x20);                              /* Issue non-specific EOI */
}

void InstallIntDHandler()
{
OrigIntDVector = getvect(0x0D);             /* Get original interrupt vector */
disable();                                            /* Disable interrupts */
setvect(0x0D,IntDHandler);             /* Point int 0Dh vector to IntDHandler */
OrigIMR = inportb(0x21);                                /* Get original IMR */
outportb(0x21,(OrigIMR | 0xDF));     /* Load new IMR value to enable int 0Dh */
enable();                                              /* Enable interrupts */
}

void RestoreIntDHandler()
{
disable();                                            /* Disable interrupts */
outportb(0x21,OrigIMR);                             /* Restore original IMR */
setvect(0x0D,OrigIntDVector);              /* Resore original int 0Dh vector */
enable();                                              /* Enable interrupts */
}
```

to access global variables. Other compiled languages, such as Pascal, support similar interrupt-type functions or procedures. Depending upon your compiler it may be necessary to disable stack-overflow checking when using interrupt functions.

The `InstallIntDHandler()` function installs the new interrupt handler by changing the interrupt 0Dh vector. It then modifies the PIC's IMR in order to enable the corresponding IRQ level. The `RestoreIntDHandler()` function effectively removes the handler by restoring the IMR and interrupt vector to their original states. The interrupt handler itself, `IntDHandler()`, is very simple. After any necessary processing has been completed, it just issues a non-specific EOI command and terminates.

Chained interrupts

So far we have seen how an independent interrupt handler can be installed on its own dedicated interrupt vector. In this scenario, the new handler completely replaces any previous interrupt handler. However, there are some cases where, although a new interrupt handler is required, the functionality of an existing handler must also be retained. It is then necessary to call the original interrupt routine whenever the new handler is invoked. In fact, it is possible to install a series of handlers on the same interrupt vector. The newest handler gets control first, performs whatever processing may be necessary and then calls the previous handler. This handler then calls the next one in the chain and so on until all handlers have been executed.

The chaining technique is widely used on the PC and is extremely useful in a variety of circumstances. You will need to chain interrupt handlers if you wish to add extra functionality to the system's timer or keyboard interrupts, for example. These are both external hardware interrupts, but software interrupts can also be chained in order to provide a means of communicating between applications programs and memory-resident driver software. The C language provides two methods of interrupt chaining: the `_chain_intr()` function and direct calls.

The _chain_intr() C function

This function is supported by Microsoft C and later versions of Borland's Turbo C. It takes, as a parameter, a far pointer to the previous interrupt handler (i.e. the one which is to be chained to). The `_chain_intr()` function may be called only from within an interrupt-type function. When `_chain_intr()` is invoked, it restores all of the processor's registers from the values previously saved on

the stack (removing them from the stack in the process) and passes control directly to the old interrupt handler. The old handler then executes as though it had been invoked directly. When the old handler has completed its processing, it returns with an IRET directly to the interrupted code – i.e. it does not return control to the new handler. The following code fragment illustrates this technique.

```
void interrupt (*OldIntHandler)();        /* Storage for original int vector */

void interrupt NewIntHandler()
{
/* Do interrupt processing here */
_chain_intr(OldIntHandler);                /* This function does not return */
/* Code here will never be executed! */
}
```

Some languages such as Pascal (and some early C compilers) do not include a `_chain_intr()` or similar function. In these cases it will be necessary to resort to assembly language programming or at least to use inline opcodes. For the benefit of Pascal programmers, the following inline macro performs a similar service to C's `_chain_intr()` function. It assumes that, on entry to the new interrupt handler, the registers are pushed in the order AX, BX, CX, DX, SI, DI, DS, ES, BP and that a stack frame is then set up by copying SP to BP (as is the case with Borland/Turbo Pascal compilers). Readers using C compilers that do not support `_chain_intr()` may wish to adopt a similar technique. If you try this, remember to account for the different order in which your compiler might save the registers on entry to the interrupt handler.

```
Procedure ChainIntr(OldIntHandler: pointer);
Inline($5B/             { POP   BX          ; Get OldIntHandler pointer      }
       $58/             { POP   AX          ;   from top of stack            }
       $87/$5E/$0E/     { XCHG  BX,[BP+OE]  ; Insert OldIntHandler in stack  }
       $87/$46/$10/     { XCHG  AX,[BP+10]  ;   at "return address" posn.    }
       $89/$EC/         { MOV   SP,BP       ; Simulate Pascal exit code by   }
       $5D/             { POP   BP          ;   restoring all registers      }
       $07/             { POP   ES          ;   from the stack. When this    }
       $1F/             { POP   DS          ;   has been completed, the      }
       $5F/             { POP   DI          ;   next two words on the top    }
       $5E/             { POP   SI          ;   of the stack are the new     }
       $5A/             { POP   DX          ;   "return addr": OldIntHandler }
       $59/             { POP   CX          ;                                }
       $CB);            { RETF              ;   "Return" to OldIntHandler    }
```

Chaining with a direct call

If you need to carry out interrupt processing *after* the old interrupt handler has been executed, your new interrupt handler will have to

call the old handler directly. The interrupt call to the old handler can be simulated by pushing the Flags register and then issuing a far call. Note that this does not simulate an interrupt exactly (i.e. it does not clear the processor's Interrupt or Trap flags), so appropriate allowances must be made. This technique can be implemented in C as follows.

```
void interrupt (*OldIntHandler)(); /* Storage for original int vector       */

void interrupt NewIntHandler()
{
/* Do interrupt processing here */
(*OldIntHandler)();                /* SAME AS: pushf                         */
                                   /*          call  DWORD PTR OldIntHandler */
/* Do further processing here */
}
```

Note that the direct call technique does not restore the registers or stack to their original state before passing control to the old interrupt handler. This is an important consideration when dealing with chained software interrupts, as most software interrupt handlers expect to receive certain values in the registers. In this case you must ensure that the new handler restores the original register contents before calling the old interrupt handler. When the old handler exits via its IRET instruction, control is returned directly to the new interrupt handler, allowing the latter to perform further processing before finally returning to the interrupted code.

Chaining hardware interrupt handlers

Because data cannot be passed via registers to an interrupt handler that is entered asynchronously, it is generally unnecessary to pass the original register contents down along a chain of *hardware* interrupt handlers. In this case the direct call chaining technique may be used. Listing 5.3 illustrates how an additional handler can be chained onto interrupt 8 (the system timer interrupt) using this technique. It is very similar to Listing 5.2, but there are three important differences. First, the new interrupt handler invokes the previous interrupt handler when it has completed its own processing. Second, because the old interrupt handler will issue the required EOI command, the new handler does not need to do this (you will need to issue an EOI if your routine does not pass control to the previous interrupt handler, however). Finally, the installation and deinstallation routines do not modify the PIC's IMR because the required interrupt level would already have been enabled by the BIOS.

Listing 5.3 *Chaining an interrupt handler on to interrupt 08h*

```
#include <dos.h>

void interrupt (*OrigInt8Vector)();     /* Storage for original int 8 vector */
         :
         :
/* Function Prototypes */
void InstallInt8Handler(void);
void RestoreInt8Handler(void);
         :
         :
void interrupt Int8Handler()
{
/* Do interrupt processing here */
(*OrigInt8Vector)();                          /* Call original int 8 handler */
}

void InstallInt8Handler()
{
OrigInt8Vector = getvect(0x08);              /* Get original interupt vector */
setvect(0x08,Int8Handler);                    /* Point vector to Int8Handler */
}

void RestoreInt8Handler()
{
setvect(0x08,OrigInt8Vector);          /* Restore original interrupt vector */
}
```

5.6 Re-entrancy and accessing shared resources

We have already noted some of the problems inherent in sharing resources between interrupt routines and non-interrupt code. If an interrupt occurs while a program is accessing a shared hardware device, and the interrupt handler then attempts to manipulate the same hardware, it is likely that this will affect the status of the device and so disturb the operation of the interrupted code. A similar consideration applies when two or more asynchronous processes need to call shared operating system services.

Any software routine that can be interrupted and then safely called again from within an interrupt handler is known as a re-entrant routine. Most DOS services are non-re-entrant and for this reason they should not normally be called from within an interrupt handler. Some BIOS services are also non-re-entrant. Fortunately there are techniques which allow access to certain DOS services from within an interrupt handler. These work by checking DOS to discover whether one of its services was being executed at the time that the interrupt occurred. Only if DOS had not been interrupted is it safe to access a DOS service from within the interrupt handler.

Further information may be found in the texts by Dettmann and Johnson (1992) and Schulman *et al.* (1990).

It should be noted at this point that the re-entrancy issue is less problematic in multitasking operating systems and real-time versions of DOS that are used in embedded PC applications. These support a number of re-entrant services which can be called from within interrupt handlers.

It is not just operating system calls that can present re-entrancy problems. You should be careful to avoid calling *any* non-re-entrant code from within an interrupt handler. This includes some driver services and routines contained within your own program. Suppose that an interrupt handler issues a call to a non-re-entrant subroutine. If your program (or another interrupt handler or task) happened to be executing that subroutine at the time of the interrupt, it is likely that the subroutine's internal data structures will have been corrupted by the time that control returns to the interrupted process.

To make a routine re-entrant it is necessary to ensure that all data structures used within the routine are dynamically allocated from a pool of free memory whenever the routine is entered. This prevents corruption of any data that might have been in use when the routine was interrupted. The most common way to accomplish this is to allocate space for new local variables on the stack each time that the routine is called. Global variables must, of course, be avoided as there can only ever be a single copy of each global variable. Care must also be exercised when accessing any other global resources, such as an item of hardware which is shared with other software subsystems. If it is necessary for an interrupt handler to access any shared device or data structure, steps must be taken to ensure that the handler can never be invoked (e.g. by disabling interrupts) while other sections of code (i.e. critical sections) are also accessing the same resource.

Re-entrancy is an issue not just for interrupt handling, but also in the design of multitasking systems. Windows NT, for example, employs a pre-emptive task scheduler that can switch between tasks or threads more or less independently of the state of the current thread. Resource conflicts are avoided by the use of re-entrant code, mutexes, semaphores and other sophisticated mechanisms built into the operating system.

5.7 Interrupt response times

The presence of asynchronous interrupts disturbs the continuous flow of a program. Hardware interrupt handlers can often cause

execution of the underlying process to be suspended for several hundred microseconds at a time. As most DA&C applications include portions of time-critical code, this disturbance can be problematic. If you use time-critical code in the non-interrupt portion of your software, you will have to either disable interrupts during execution of the code (which is practicable over only short time intervals) or be prepared for the code to be interrupted at unpredictable intervals.

A more satisfactory alternative is to place the important code within an interrupt handler. This has two advantages. First, the routine will only be executed when it is needed: the software will not have to perform continual checks to determine when the code should be activated. Second, if priorities are carefully assigned, the interrupt handler will also be less likely to be interrupted itself.

A certain amount of overhead is always involved in responding to an interrupt and transferring control to and from the associated interrupt handler. This can often result in a lower throughput than if a non-interrupt polling loop is used. As well as limiting the rate at which I/O and other operations can be performed, the interrupt overhead also *delays* the response of the system to individual interrupt requests.

At this point it should be noted that interrupt sharing, which is possible on MCA systems, can introduce a small but potentially significant additional overhead because the interrupt handler has to determine which of the attached devices requires service. Sharing an interrupt line between two (or more) subsystems should be avoided in situations where the fastest possible interrupt response is required.

The time taken to respond to an interrupt request (i.e. to perform some useful action) is determined by two components: the interrupt latency time and the speed at which the interrupt handler itself performs its allotted task. The latter is dependent upon the nature of the application and is often relatively easy to optimize by adopting efficient coding practices. The interrupt latency time, on the other hand, is much more difficult to quantify or control. It represents the *worst-case* time taken for the system to respond to an interrupt request. It is defined as the maximum interval between the point in time where the interrupt request is asserted and the instant that the processor commences execution of the associated interrupt handler. The interrupt latency time is composed of three elements:

1. The interrupt recognition time (T_R).
2. The time required to complete the current instruction (T_I).
3. The interrupt processing time (T_P).

T_R is the time taken by the processor to recognize that the interrupt request is pending. If interrupts have been masked by means of

a CLI instruction, or temporarily disabled at the PIC, T_R can be quite considerable. Unfortunately, it is not always easy to determine how long the system keeps interrupts disabled. Device drivers and operating system services, which the program might have cause to invoke, may disable interrupts for an indeterminate length of time. The time during which an interrupt may be blocked by a higher priority routine can contribute significantly to its latency time. All possible combinations of interrupts occurring at the same time (or nearly the same time) must be taken into account when assessing the worst-case value of T_R. Certain instructions can also temporarily mask interrupts. We have already mentioned the STI and IRET instructions which do not allow interrupts to be enabled until after the next instruction has been executed.

In addition, the processor disables interrupts between LOCK and segment-override prefixes and the instructions to which they relate. Instructions which modify the contents of the segment registers on the 8086 and 8088 processors also cause interrupts to be disabled until after the following instruction has been completed. However, this only applies to instructions which modify the SS register on 80286 and later processors. The occurrence of higher priority interrupts can also increase T_R by preventing lower priority handlers from executing for perhaps several hundred microseconds, or more.

The second component of the interrupt latency time, T_I, depends on the nature of the instruction that is being executed at the time the processor detects the interrupt. Most instructions take a few microseconds to execute on an 8088 processor (often much less than 1 μs on more modern systems). However, some operations such as multiply or divide may take approximately five or ten times longer to execute.

The interrupt processing time (T_P) is usually of less significance than T_R, although it is an important factor in determining the minimum possible interrupt latency. It represents the time taken by the processor, after it has recognized the interrupt request, to acknowledge the interrupt (i.e. to issue the necessary INTA cycles), save the Flags, CS and IP registers, retrieve the interrupt vector and transfer control to the interrupt handler. For external hardware interrupts on a 4.77 MHz 8088-based machine, this procedure takes approximately 12.7 μs. A slightly shorter processing time is required for an NMI: typically 10 to 11 μs on an 8088 processor. Later processors running at higher clock speeds are, of course, able to perform the same operations in considerably less time.

In order to calculate the interrupt latency time, the worst-case values for T_R, T_I and T_P must be added together. In most applications T_R is by far the most important contributor to the interrupt

latency time. Nevertheless, it can be a difficult task to determine the maximum value of just this one quantity, particularly on systems running DOS or Microsoft Windows, which were not designed specifically to meet the stringent timing requirements of real-time applications.

Chaining of interrupt handlers can further complicate the problem, making interrupt latencies more difficult to predict. This is especially so if you have no control of what other software the end user may install on the same interrupt.

The programmer must always ensure that the interrupt response of the system is adequate regardless of what portion of the software is being executed. Consideration should be given to the effect on interrupt latency of all sections of code in the system. This includes critical code sections (i.e. code executed with interrupts disabled), calls to operating-system (and BIOS) services and execution of other interrupt handlers.

In DOS and Windows-based systems, one largely unknown quantity (and one over which the programmer has little control) is the interrupt latency introduced as a result of operating-system code. Often, there is little information available on interrupt masking within the various system services. In addition, task and mode switches under Windows can make interrupt latencies much more difficult to predict.

Details such as this tend to be precisely quantified in specialist real-time operating systems. These include ROMable versions of DOS and the BIOS which are widely used in embedded PC systems. They are designed for use in multitasking real-time environments, offering well-defined interrupt latencies, and are essential if the PC is to be used for high speed real-time applications. They are also often (at least partially) re-entrant and this allows operating-system services to be called from within interrupt handlers.

Interrupt latencies are, generally speaking, greatest for systems running under Microsoft Windows and those executing in protected mode under a DPMI server. In these systems, calls to operating-system services may involve switching the processor from protected mode to real (or V86) mode and then back again. Mode switches as well as task switches are frequently necessary in order to service hardware interrupts. Whether a mode switch occurs depends upon the mode of the processor at the time of the interrupt and whether a suitable interrupt handler exists for that mode. In normal operation, Windows 3.1 might perform, perhaps, 20 or more mode switches every second. Mode switches can be quite time consuming (a few microseconds up to a few hundred microseconds on an 80286 processor) and unless great care is taken they can severely degrade the system's

real-time performance. When Windows is running several processes concurrently, interrupt requests have to be routed to the appropriate thread or task in order for them to be handled properly. The time required for this routing and consequent context switches is variable and depends upon many factors. This can make it very difficult to predict interrupt response times under Windows. Whatever operating system is used, careful design and a detailed knowledge of the peculiarities of the operating system are of paramount importance in assessing the interrupt performance and real-time characteristics of the system.

In many applications the real response to an event does not occur until after the interrupt handler has terminated. The handler may, for example, only transfer data to a buffer or set flags: the data or flags are then acted upon by another portion of the software (e.g. a loop within the *interrupted* process or, in the case of a real-time multitasking system, by a related task). The response of the software as a whole (e.g. the loop cycle time or the time required to invoke the task) will then determine the actual performance of the system.

Often the only feasible course of action is to determine the overall response of the system by thorough and exhaustive testing. Bear in mind that the actual latency time measured empirically for any one interrupt may not be representative of the worst-case interrupt latency. This figure is often difficult to measure because hardware interrupt processes are, by their nature, asynchronous. This means that interrupt requests can occur while the system is in almost *any* state and it may, therefore, be impracticable to reproduce all possible combinations of interrupts and system conditions during testing.

6 Data transfer

We will now turn our attention to a topic of central importance in data acquisition and control: transferring data between the PC and a peripheral DA&C device. The data transfer techniques that can be adopted in a DA&C program will depend, to a great extent, upon the nature of the DA&C hardware to be used. This chapter introduces the types of device that are available for interfacing to DA&C systems and discusses a number of issues and software techniques related to data transfer. The following two chapters continue this theme, covering parallel and serial buses and associated devices in more detail.

6.1 Data-acquisition interface devices

By a DA&C interface device, I mean a device that facilitates connection of sensors and actuators to the PC. These take many different forms. It is convenient to classify them according to their processing capability and the way in which they transfer data to and from the PC. These considerations govern how the software communicates with the DA&C device and determine, to a great extent, the internal structure and capabilities of the software.

In the following discussion I will use the terms 'intelligent' and 'dumb' to refer, respectively, to programmable devices that are able to autonomously process and manipulate acquired data, and to devices that possess no such processing capability. These informal terms are used only for convenience. This usage is somewhat imprecise and does not, of course, indicate the presence, or otherwise, of any form of artificial intelligence.

Connection to the PC

The simplest DA&C interface devices consist of circuit boards that are plugged directly into the PC's system-bus (e.g. ISA or PCI) expansion

sockets. These devices each provide one or more hardware registers that are mapped into the PC's memory or I/O space. Because such devices connect directly to the system bus, data can be transferred between the device and software in one operation. For example, a simple assembly language OUT instruction might be all that is required to change the state of a group of eight digital output lines or relays.

Communication with intelligent devices involves an intermediate step. They buffer and translate command codes sent via the registers and then act on the command, transmitting the appropriate digital bit patterns to the ADC, relays or to other interface components.

Although plug-in interface cards are the cheapest and, perhaps, the most widely used interfacing solution, they are not practicable if, for example, sensors are to be located at a remote site. Where signal losses preclude the use of long sensor leads, the PC and digitizing device may have to be positioned some distance apart. In these situations an external serial link or parallel bus (e.g. RS-232, RS-485 or IEEE-488) will usually be required to carry commands and digitized signals between the PC and a remote DA&C unit. Interfacing techniques for serial and parallel buses are discussed in Chapters 7 and 8.

Intelligent DA&C devices

Devices that possess a degree of on-board intelligence may assume a number of data collection, storage and processing tasks which would otherwise have to be undertaken by the PC. These devices are usually designed to facilitate deterministic operation and provide guaranteed response times and data-acquisition rates. Such capabilities can obviate the need for complex deterministic and/or multitasking PC operating systems and can often help to simplify the DA&C software. A dedicated on-board processor may, for example, be programmed to execute a deterministic control algorithm while leaving the PC free to perform other tasks (e.g. to manage the user interface or to provide disk storage).

It is often somewhat simpler to communicate with intelligent DA&C devices than to directly manipulate the control lines and registers of dumb I/O cards. The PC programmer does not have to be aware of how the various DA&C subsystems (e.g. ADC, multiplexer, sample and hold) function; all that needs to be understood are the end results of issuing particular high level commands to the device's microcontroller. These commands may be used to configure the device or to initiate simple tasks such as reading an analogue input channel. They may also perform more complex operations such

as programmed scanning of multiple input channels, buffering acquired data or even scaling and linearizing each reading.

High level command sets offered by most devices are both simple and flexible, but they do introduce an additional layer of complexity between the PC and the low level data-acquisition hardware. Depending upon the nature of the device, the PC software may have to accommodate a more complex communication protocol – particularly in the case of serial bus devices (see Chapter 8). The extra processing required to formulate, issue and interpret commands may in some applications limit the speed and efficiency of the system as a whole.

An important characteristic of some intelligent DA&C units is the ability to transmit data to the host PC in the form of ASCII encoded character strings. This permits both scaled and unscaled data to be transferred. Many devices take advantage of such a capability by providing facilities for on-board scaling or linearization of data. The capacity to scale acquired data allows the device to support a number of more advanced features, such as the ability to operate as an autonomous controller, to respond to trigger events or to record only data that falls outside predefined limits. The penalty paid for these facilities is, in many cases, significantly reduced throughput.

Plug-in coprocessor and DSP cards

One of the simplest solutions for DA&C applications that require intelligent I/O is to employ a plug-in coprocessor card. These are simply single-board computers that are designed specifically for data acquisition, analysis and control. The DA&C coprocessor can be programmed to perform all of the time-critical operations. As the host PC is normally used only in a supervisory role and/or to supply mass storage, user I/O and peripheral interfacing facilities, its performance is normally not critical. This type of system is particularly suited to computationally intensive tasks where acquired data must be mathematically processed in real time. Typical examples include audio signal and vibration analysis and a variety of real-time process-control applications. Although most coprocessor cards do not incorporate analogue signal conditioning (to minimize wide-band noise pickup from the digital circuitry), many possess a number of ADC channels, DACs, digital I/O ports and timers.

80x86 coprocessor cards

A small number of coprocessor cards are based upon the 80x86 family of microprocessors and have an architecture similar to that of the PC. They are suited to a wide range of real-time DA&C applications and usually permit high speed operation, with maximum

sampling rates ranging from about 50 to 300 000 samples/s. With suitable buffering, some cards can stream data directly to the host PC's hard disk at rates up to about 100 KB/s. These devices are often equipped with a moderate amount of system RAM. Some also include dedicated FIFO memory buffers to facilitate high speed data capture. A few models will operate in PC and XT class machines, but most require an AT compatible (ISA) bus slot.

The I/O facilities offered usually include high speed analogue inputs, analogue outputs and digital I/O lines. Some manufacturers supply modular boards which can be tailored to specific applications by adding additional ADCs, DACs or digital I/O ports. Direct memory access (as described later in this chapter) is often supported, together with flexible interrupt and timing systems.

Some cards have their own ROM-based real-time operating systems (RTOSs). These provide dedicated DA&C functions and facilitate communication with the host PC. Special drivers and development utilities are usually supplied with these systems, allowing data-acquisition, data-processing and control algorithms to be downloaded to the target processor. Depending upon the type of processor and operating system used, these programs may be in executable form or may be written in a specialized script language that is interpreted by the RTOS.

Digital signal processors

80x86-based cards are suitable for a variety of DA&C tasks, but for high speed signal-processing applications a specialized Digital Signal Processor (DSP) is generally a more satisfactory alternative. A DSP is essentially a microprocessor that is optimized for running numerically intensive signal-processing algorithms. Key features of such systems are high accuracy and, in most cases, very high rates of throughput. A number of manufacturers supply ISA cards equipped with one or more DSP chips. At least one presently provides a DSP card for the PCI bus. A number of DSP-equipped PCMCIA cards are also now becoming available for notebook computers.

As well as allowing the PC's processor to execute concurrently with the DSP, a plug-in DSP card can itself form the basis of an inherently parallel architecture. Some implementations permit multiple DSPs to be connected together in a variety of powerful parallel-processing topologies. Each DSP can be programmed to execute different signal-processing functions or to perform the same processing on different sets of data. This inherent parallelism means that DSP cards are ideal platforms for real-time applications or when large arrays of data have to be processed.

DSPs can be programmed to execute a variety of high speed data-processing and control algorithms. Some of the most common are signal comparison, fast Fourier transforms, convolution, frequency measurement, scaling, linearization, statistical functions, waveform synthesis, PID control and digital filtering. Many of these can also be performed by the PC itself (albeit somewhat less efficiently) and these are discussed at various points throughout this book. Typical DSP applications include vibration analysis, machine condition monitoring, spectral analysis, audio frequency applications, engine analysis, digital image processing and high speed real-time control. DSPs are also often used in embedded systems. In these cases, the PC is used only as a convenient platform for development of DSP code and takes no part in the actual data acquisition.

The I/O facilities provided by DSP coprocessor cards tend to vary between different models, but most are equipped with between one and 16 high speed analogue input channels and a number of analogue outputs, digital I/O ports and timers. FIFO memory buffers are often used to decouple the digitization and DSP circuitry. They usually possess flexible interrupt and DMA (Direct Memory Access) systems, which support high speed transfer of data to the host PC. Data transfer is facilitated on some cards via a block of dual-ported RAM mapped into the PC's memory space.

DSP cards are normally controlled via on-board firmware. This includes DSP libraries that contain commonly used algorithms. Many manufacturers also provide complete software development environments (including an assembler, compiler and debugging software). Source files are edited and compiled on the PC and the executable software is then downloaded to the DSP card. Library functions may also be included to allow access to the host PC's console and I/O facilities.

Remote DA&C units

Most remote DA&C units are capable of some degree of independent processing. These devices generally incorporate dedicated microcontrollers and possess their own ROM-based operating systems. Many allow moderately high speed operation, although the degree of determinism that they offer does tend to vary between different models. Because of their autonomous processing and data-storage capabilities they are often used for stand-alone data logging and control. Facilities for analogue and digital output may be supplemented by software comparators or control algorithms. These can help to relieve the less deterministic PC of the burden of real-time control: a considerable benefit to the DA&C programmer. There

are three main classes of remote DA&C unit (as well as many hybrid devices):

1. Single-channel I/O units are usually connected to the PC via a multi-drop network such as RS-485. These devices are commonly used where many sensors have to be widely distributed over a large structure such as a bridge or dam. In these cases there are usually numerous devices attached to a single network. Each unit or I/O channel is usually addressed by means of a unique identification code. This type of device frequently has only a limited capacity for on-board buffering or data processing.
2. Multi-channel data loggers are normally connected to the PC on a one-to-one basis via a serial or parallel interface. Most devices possesses at least eight or 16 analogue input channels. This may be expandable up to several hundred channels on some systems. A numeric code is assigned to each I/O channel and the software must use this code in order to configure that channel or to read data from it. Many of these devices have quite sophisticated processing abilities. Some are able to buffer large quantities of data, to store data on disk drives or to interface to modems, printers or plotters. For this reason they are often used for stand-alone data logging and may only need to be connected to the PC for programming or to download acquired data.
3. Stand-alone laboratory instruments and test equipment can also, in many cases, be interfaced to the PC for data acquisition. Most of these instruments have a degree of intelligence and are capable of periods of independent operation. Many are designed for specialized test and measurement work and the facilities which they provide are often tailored to specific applications such as spectrometry, pH sensing, chromatography or audio frequency analysis. The RS-232 or IEEE-488 buses (see Chapters 7 and 8) are normally used for interfacing to this type of device.

Most remote DA&C devices possess the signal-conditioning circuitry necessary to interface to sensors and/or actuators. They often have a modular construction, which allows the end user to select the appropriate type of analogue signal-conditioning unit and/or digital-I/O interface. In this way the system is able to accommodate various types of sensor (e.g. thermocouples, strain gauges, or LVDTs) as well as relays and opto-isolated digital I/O devices. The PC software may have to support all possible configurations and may need to interrogate the DA&C unit to determine which modules are installed.

Dumb interface devices

Many simple analogue or digital I/O cards that connect directly to the ISA bus, PCI bus or PCMCIA slot have little or no on-board processing capability. Instead, virtually all aspects of the interface device's operation are controlled by the PC via I/O-mapped or memory-mapped registers. The PC initiates data transfer and manages the flow of data across the interface. These duties can be quite processor intensive, particularly where many I/O channels and high sampling rates are involved.

Although directly manipulating the registers and control lines of plug-in cards can be somewhat more involved than communicating with an intelligent DA&C unit, such an arrangement often provides a greater degree of control over the data-acquisition process. Because the PC is usually responsible for managing each component of the device, there is generally much more scope for varying the timing and order of channel selection, sample-and-hold triggering, gain selection and ADC reading operations. For this reason the data-acquisition process can, in some circumstances, be carried out more efficiently than would be possible using an intelligent DA&C unit.

The fact that the PC's software is responsible for all aspects of the data collection and control operations can also be a serious disadvantage. If you are working to a tight timing specification, it may be necessary to adopt a specialized real-time operating system and to dispense with any non-deterministic, but otherwise desirable, features of the software. You should also bear in mind that when directly manipulating registers and control lines there is a greater potential for software errors to find their way into your DA&C program. These can be quite subtle and time dependent. They may not become apparent during static testing, only showing themselves at high rates of throughput, on certain high speed models of PC or when a specific sequence of events occurs. Time-dependent software errors can be very difficult to reproduce and trace during testing.

6.2 Data transfer techniques and protocols

There is usually no inherent synchronization between DA&C hardware and the software running on the PC. Components such as ADCs and multiplexers are said to operate asynchronously with the PC. In such a system, it is not possible to predict the state of the DA&C hardware at any particular time and the PC must, therefore, have some way of determining whether a peripheral device is busy or whether it is safe to access it. In order to ensure that data is not

presented to the PC at too fast a rate (and, conversely, to prevent the PC from demanding data at too fast a rate) it is essential to establish a set of rules, or protocol, for data transfer.

Handshaking

In the case of a simple plug-in ADC card, it is usually necessary to initiate analogue-to-digital conversion and then wait until the conversion is complete in order that valid data can be read from the ADC. We have seen in Chapter 3 that this requirement can be implemented by a handshaking protocol that uses the ADC's Start Conversion (SC) and End of Conversion (EOC) control lines. Intelligent DA&C devices, which often communicate with the PC via a serial link (e.g. RS-232, RS-485 etc.), must also operate in accordance with a strict communication protocol.

Protocols are usually effected by means of handshaking or control signals that indicate the state of readiness (or otherwise) of some element of a device. These signals are usually transmitted via digital I/O lines (e.g. an ADC's SC and EOC lines). Other types of I/O interface employ slightly more complex handshaking techniques, but the basic principle is the same: to facilitate an orderly, synchronized transfer of data.

Many serial communications systems provide for an alternative protocol known as software handshaking or character flow control. Installations that do not use the serial port's handshaking lines can transmit special control characters to regulate the flow of data along the serial bus. This technique is described in more detail in Chapter 8.

Data I/O strategies

The protocols involved in communicating with any DA&C device will, of course, depend upon the nature of the communications interface employed (e.g. serial or parallel bus or direct connection to the PC's expansion bus) and upon the degree of synchronization inherent between the PC and the device. Because communications mechanisms and protocols vary considerably, it is not appropriate to discuss details of specific devices here (although certain standard protocols and handshaking techniques for use with parallel and serial bus-based systems are discussed in Chapters 7 and 8). Of more general interest are the strategies that you can adopt within your data-acquisition programs for requesting and receiving data from DA&C devices. What follows applies, in general, to both intelligent

and dumb DA&C devices, although the details of the mechanisms involved will, of course, be somewhat different in each case.

Input

The simplest technique for inputting data from a device is to configure it so that it operates in a free-running mode, providing data at its fastest possible rate. The software can then periodically poll the device to detect whether it has new data. An example of this is the free-running ADC technique in which the ADC's End of Conversion (EOC) output is connected (if necessary, via suitable logic) to its own Start Conversion (SC) input. This results in continuous analogue-to-digital conversion. The PC software monitors the EOC signal to detect when the ADC has completed each conversion and then reads the new digitized value from the ADC's output buffer.

Alternatively the DA&C device (i.e. ADC or intelligent data logger) may be configured to take readings at regular intervals under the control of a hardware timer. This technique is useful where readings are to be taken at precise intervals. From the software's point of view, it is similar, in principle, to the free-running technique. Both approaches free the software from having to decide when to initiate analogue-to-digital conversions. They do, however, require the DA&C program to be ready to respond at any time that new data is made available.

Other techniques give the software more control over the timing of the data-input process. The PC software may be designed to request data either by issuing a suitable high level command or by outputting an SC signal to an ADC. The timing of a data-request command may be controlled in several ways. The software might request a new reading as soon as previous data has been processed; when it detects user input (e.g. a key press or mouse click); on receipt of digital handshaking signals from other components of the DA&C system; or by reference to an elapsed time timer. In the latter three cases it is possible (and often preferable) to issue data request commands from within a hardware interrupt handler.

Data may not always be immediately available after the PC has requested a new reading. The software will generally have to wait (or continue with some other task) while the DA&C interface device interprets the command, selects the appropriate input channel, or digitizes and processes (e.g. scales or linearizes) the data. The DA&C program must incorporate some mechanism for determining when valid data is available. The software may poll a designated I/O port in order to determine the state of a 'data available' flag. Alternatively, a handshaking signal could be fed to an IRQ line in order to generate an interrupt whenever the DA&C device wishes to transmit new data.

The interrupt handler may then read the acquired data or it may just set a flag to cause the main data-acquisition routine to read the data when interrupt processing has been completed.

Summary

The following strategies are available for determining when to request data or for initiating ADC conversions:

1. Polling. Software or hardware flags may be periodically checked from within a software loop in order to determine when the system is ready to supply and/or process more data. The state of these flags may be controlled via user input, digital control inputs or *software* timers.
2. Hardware interrupts. Interrupt handlers for system timers, user-input devices, serial/parallel ports or other peripheral devices are often convenient locations for code which initiates or manages I/O operations. The software is free to perform other tasks when not processing interrupts.
3. Direct hardware control. Hardware devices such as simple counter/timer circuits can be configured to periodically initiate actions such as analogue-to-digital conversion or to control the timing of handshaking signals.

The software may subsequently detect and read new data, either by polling the DA&C device or by installing interrupt handlers that respond whenever new data becomes available. DA&C devices that continuously transmit a stream of data without any form of handshaking (e.g. some RS-232 systems) will generally require the software to employ an interrupt-driven input mechanism in order to ensure that no data is lost.

Output

Outputting analogue data often involves only a single write operation to an I/O port. For this reason it is usually more straightforward than inputting analogue data which normally requires a two-stage ‘request and read’ operation. However, the system must regulate the flow of output data, which is normally accomplished by means of handshaking signals (in addition to any high level communications protocols that may be required). These may be used to strobe data out to a peripheral device, thus allowing outputs to be updated only when it is safe to do so. Handshaking may be implemented using digital I/O control lines or via high level commands or status polling facilities (depending upon the nature of the DA&C device). Both polling and interrupt-based techniques can be used for sensing handshaking signals and for managing data output.

Comparison of interrupt and polled I/O

We have seen that there are two techniques at the programmer's disposal that can be used for sensing the state of handshaking signals: polling or interrupts. Each method has its own particular advantages and disadvantages. Which is most appropriate will depend upon the nature and structure of your application. This section provides some general guidance.

Polling is the most straightforward technique. It simply involves reading the state of a digital I/O line via either an I/O-mapped or memory-mapped register. This is done by means of an IN or MOV instruction or high level language counterpart. Polling can be performed in a data-acquisition software loop together with any other operations that may be necessary. Alternatively, a dedicated polling loop can be used. In this case, the handshaking line or flag is repeatedly checked until it changes state, at which point the loop is terminated and control is passed to an appropriate routine. Efficient polling loops written in assembly language – such as that illustrated in the following code fragment – can provide a very rapid response to changes in the state of handshaking lines or other digital inputs.

```
            mov   dx,300h      ;I/O Port address to read
            mov   bl,80h       ;Mask to select bit 7 of input byte
LoopStart:  in    al,dx        ;Read port
            test  al,bl        ;Select status bit (i.e. bit 7)
            jz    LoopStart    ;Loop if status bit = 0
             :      :
            ;Status bit = 1 so perform
            ;any necessary processing here
             :      :
```

Interrupts can also provide a rapid response, but because of the overhead involved in recognizing an interrupt, invoking the interrupt handler, acknowledging the 8259 PIC and then transferring control back to the interrupted process (see Chapter 5) the maximum throughput achievable is often lower than if a well-written polling loop were to be used.

As well as limiting the rate at which I/O operations can be performed, the interrupt overhead also *delays* the response of the system to individual interrupt requests. The overheads inherent in managing interrupts can mean that timing precision is often much worse (by a factor of *at least* 5 to 10) than if using a polling loop. For reasons outlined in Chapter 5, interrupt response times are variable and often relatively long. Depending upon the operating system used, they may also be indeterminate. This is an important

consideration when writing software that must respond quickly to time-critical events.

In spite of their less efficient response times, interrupts provide a number of very important advantages over polling. First, they allow the software to continue with other tasks instead of simply waiting for input. The more efficient use of available processor cycles can often compensate for the inefficiencies inherent in responding to individual interrupts, improving the overall throughput. An interrupt-based event-driven I/O system also permits a more modular software structure to be employed, and this can go some way to improving the reliability of the DA&C program.

Memory- and I/O-mapped transfers

Whether data acquisition is performed via a serial link, external parallel bus or via a DA&C card connected directly to the PC's expansion bus, all I/O operations are ultimately performed via registers mapped to either the PC's memory or I/O space.

In the memory-mapping scheme, control registers and I/O latches are assigned to one or more (usually contiguous) memory locations. These are often within the PC's 1 MB real-mode addressable region: particularly in the upper memory region between 640 KB and 1 MB. Hardware designed for use with 32-bit processors and operating systems may use other physical memory addresses up to 4 GB. Data is transferred to and from memory-mapped registers by simply reading or writing the appropriate memory address. Memory-mapped I/O is not widely used on PC adaptor cards.

The majority of data-acquisition interface products possess a group of (typically 4, 8 or 16) control and data registers, and these are mapped to a configurable address range within the PC's I/O space. The registers may be accessed using assembly language `IN` or `OUT` instructions or their high level language counterparts. Although a detailed discussion of programming languages is outside the scope of this book, I/O instructions and functions are of such central importance to the subject of data acquisition that we shall briefly consider this topic below. Only three implementations are covered, but most PC programming languages provide similar facilities. There may, however, be slight differences between dialects of the same language. You should consult your programming language manual for more precise information.

Accessing I/O-mapped registers in assembly language

Assembly language provides a wealth of instructions for performing 8-, 16- and 32-bit I/O operations. All members of the 80x86

family of processors support the basic IN and OUT instructions. Newer members (i.e. 80386 and later) also support a number of string I/O instructions (i.e. INSB, INSW, INSD, OUTSB, OUTSW and OUTSD) which are very useful for transferring large quantities of data between a memory buffer and a peripheral device. The various I/O instructions are listed in Table 6.1.

IN and OUT instructions

The IN and OUT instructions have already been introduced in Chapter 1. These instructions always transfer data to or from the accumulator; no other registers can be used. 1-, 2- or (on 80386 and later systems) 4-byte transfers are allowed, depending upon whether the AL, AX or EAX register is specified. If the I/O port number is less than 100h, it can be coded as an immediate byte constant. If it is greater than or equal to 100h, the port number must be specified in the DX register. The various forms of the IN and OUT instructions are summarized in Table 6.2.

Table 6.1 *Assembly language I/O instructions*

Instruction	*Processor*	*Description*
IN	*8086+	Reads 8-, 16- or 32-bit values from the I/O ports to the accumulator.
OUT	*8086+	Writes 8-, 16- or 32-bit values to the I/O ports from the accumulator.
INSB	*80186+	Byte-by-byte string input to ES:[DI/EDI].
OUTSB	*80186+	Byte-by-byte string output from DS:[SI/ESI].
INSW	*80186+	Word-by-word string input to ES:[DI/EDI].
OUTSW	*80186+	Word-by-word string output from DS:[SI/ESI].
INSD	80386+	Dword-by-dword string input to ES:[DI/EDI].
OUTSD	80386+	Dword-by-dword string output from DS:[SI/ESI].

*80386+ required for 32-bit transfers/addressing.

Table 6.2 *The assembly language* IN *and* OUT *instructions*

Direction	*Port*	*Byte I/O*	*Word I/O*	*Double word I/O*
In	<100h	IN AL, *port*	IN AX, *port*	IN EAX, *port*
In	Any	IN AL, DX	IN AX, DX	IN EAX, DX
Out	<100h	OUT *port*, AL	OUT *port*, AX	OUT *port*, EAX
Out	Any	OUT DX, AL	OUT DX, AX	OUT DX, EAX

String I/O instructions

The string I/O instructions work in much the same way as the equivalent string move (MOVS, MOVSB, MOVSW and MOVSD) instructions. The former allow 8-bit, 16-bit or (on 80386 and later processors) 32-bit data to be transferred directly between memory and an I/O location specified in the DX register. This is a very efficient way of transferring large amounts of data between a peripheral device and a memory buffer. It is a useful alternative to Direct Memory Access (DMA) for block data transfers, although DMA can provide better throughput under some circumstances.

The INSB, INSW and INSD instructions all read data from the I/O port address specified in DX directly into the memory location addressed by ES:[DI] (or ES:[EDI] in 32-bit address mode). The DI (or EDI) register is automatically incremented or decremented, depending upon the state of the direction flag, by an amount equal to the number of bytes (i.e. 1, 2 or 4) transferred.

The OUTSB, OUTSW and OUTSD instructions complement the string input instructions. Data is written from the 1-, 2- or 4-byte memory location specified by DS:[SI] (or DS:[ESI] in 32-bit address mode). The SI (or ESI) register is automatically incremented or decremented, depending upon the state of the direction flag, by an amount equal to the number of bytes (i.e. 1, 2 or 4) transferred.

The string I/O instructions can be used in conjunction with the REP prefix to transfer a string of bytes, words or double words. The number of elements in the string is specified in the CX register (or optionally the ECX register on 80386 and later processors) as follows:

```
mov   es,SEG InputBuf      ;ES:DI --> Start of InputBuf
mov   di,OFFSET InputBuf   ;
mov   dx,PortNum           ;DX contains I/O port number
mov   cx,40h               ;CX = Number of times to repeat
cld                        ;Clear Direction Flag so DI increments
rep   insw                 ;Read string
```

The generic form, INS or OUTS, can be used instead of specifying the data size explicitly in the instruction mnemonic. If this form is used, you will have to specify the size of data to be transferred by including a BYTE PTR, WORD PTR or DWORD PTR operator in the source or destination memory reference. For example, a 16-bit string output instruction (in 16-bit address mode) could be specified either as:

```
outs     WORD PTR ds:[si],dx
```

or simply as:

```
outsw
```

Both instructions have the same effect. Similar constructs may be used in the case of the other string I/O instructions. If you use the generic INS or OUTS form, you should bear in mind one important peculiarity. As with the string manipulation (MOVS etc.) instructions, the effective address of the destination/source operand specified in the instruction is actually ignored. This operand is only used to specify the *size* of the data transfer; the actual address contained in the instruction operand does not matter. Inputs are always directed to the memory address specified by the current ES:[DI/EDI] registers, while output values are always sourced from the memory address specified by DS:[SI/ESI]. You could, for example, specify the following instruction in place of either of the preceding forms:

```
outs      WORD PTR [bx],dx
```

where the BX register contains some undefined value. The operand address governed by the contents of the BX register actually has no effect. All three of the above forms would have the same end result.

According to Hummel (1992), some versions of the 80286, 80386 and 80486 processors do not execute the string input instructions correctly under certain circumstances, particularly in protected mode. In addition to these problems, the I/O protection mechanisms used in protected and virtual-8086 modes add a number of additional complications which tend to negate the advantages offered by the string I/O instructions. It is often simplest to avoid using the string I/O instructions unless your software will run only in real mode. If you do wish to use these instructions in a protected-mode environment such as Windows, OS/2 or under a DOS extender, you should consult a text such as that referenced above for additional information.

Back-to-back I/O

Perhaps the most important potential sources of error are related to the timing of I/O operations. Many I/O registers require a short amount of recovery time after an I/O operation is performed. If, for example, two I/O operations are performed on the same I/O port in quick succession, data transferred during the second I/O port access may become corrupted. This can be particularly problematic if the string I/O instructions are used with the REP prefix, as successive repetitions of the I/O instruction occur immediately after the last operation has been performed. Some ISA systems employ hardware solutions such as inserting wait states in all I/O operations. EISA systems are designed to avoid these difficulties.

The software solution is, however, very simple and easy to implement. To make your software as immune as possible to I/O timing problems it is prudent to include a short delay immediately after each IN or OUT instruction. A safe delay period is typically of the order of 1 μs (although this figure can be variable). On slow 80486 and earlier computer systems, a few short jumps will normally suffice. For example:

```
out     dx,ax
jmp     SHORT $ + 2
jmp     SHORT $ + 2
jmp     SHORT $ + 2
in      ax,dx
```

Because the timing of JMP instructions varies between different systems, this method will result in a variable delay time. On faster machines, many JMP instructions may be needed to provide the required delay. A more robust alternative is to create a calibrated software delay loop.

Delays are not included in the examples in this book unless back-to-back I/O is performed. These examples will work satisfactorily on many systems, but you may need to add an I/O delay when accessing slow peripherals or when using a fast PC.

Timing of multiple-byte transfers

Under certain circumstances multiple-byte data transfers using I/O instructions require more than one bus cycle. The timing of data transfers is governed by the processor and type of expansion bus in use. You should be aware that more than one bus cycle may be required to transfer 2- or 4-byte data to unaligned port addresses. An unaligned address is either a group of two ports that is not aligned on a word boundary (i.e. an even address) or a group of four ports that is not aligned on an address divisible by four. The fact that more than one bus cycle is required for unaligned I/O means that data may be transferred in two or three discrete steps. The precise order with which the component ports are accessed is undefined and may vary between different systems. For this reason, it is inadvisable to use such transfers within your program if you need to retain control over the order in which the individual ports are accessed. In such cases you should code the individual port accesses explicitly, or at least use a data size small enough to ensure that only aligned I/O operations are performed.

Accessing I/O-mapped registers using a high level language

The C and C++ programming languages provide several functions and macros for reading and writing both byte- and word-sized I/O ports. There are slight differences between the Microsoft and Borland implementations as shown in Table 6.3. Note, however, that Borland C also supports the Microsoft I/O functions. In both cases, I/O is performed by calling the function whose declaration is shown in the table. Examples illustrating how Borland C can be used for accessing I/O-mapped peripheral devices are given in Chapters 7 and 8. BASIC programs also use a similar method, providing an `INP` function and `OUTP` statement. Some dialects of BASIC will support only 8-bit I/O operations.

Borland Pascal (including versions of Turbo Pascal) adopts a different, and arguably more intuitive, approach. I/O functions and macros are not used. Instead the I/O port addresses are declared as one-dimensional arrays called `Port` and `PortW`. The ports are read or written in the same way as any normal array element would be accessed, as shown in Table 6.3. The elements of the `Port` array are of type byte and those of the `PortW` array are of type word.

The delays inherent in calling high level I/O functions are usually sufficient to avoid the recovery problems that occur when performing back-to-back I/O. However, some hardware may take an unusually long time to process data and in these cases you may have to include

Table 6.3 *I/O port access from high level languages*

Language	*Direction*	*Bytes*	*Declaration/usage*
Microsoft C	In	1	`int inp(unsigned port)`
	Out	1	`int outp(unsigned port, int data)`
	In	2	`unsigned inpw(unsigned port)`
	Out	2	`unsigned outpw(unsigned port, unsigned data)`
Borland C	In	1	`unsigned char inportb(int port)`
	Out	1	`void outportb(int port, unsigned char data)`
	In	2	`int inport(int port)`
	Out	2	`void outport(int port, int data)`
Borland Pascal	In	1	`Data8 := Port[PortNum];`
	Out	1	`Port[PortNum] := Data8;`
	In	2	`Data16 := PortW[PortNum];`
	Out	2	`PortW[PortNum] := Data16;`

appropriate delay loops or other synchronization mechanisms within your code.

Direct memory access (DMA)

The processor's IN and OUT instructions are often capable of providing more than adequate rates of throughput. However, some high speed systems demand faster I/O techniques. Input instructions require data to be transferred in two stages: from the peripheral device to the accumulator (AL, AX or EAX registers) and then from the accumulator to memory. The alternative technique of Direct Memory Access (or DMA) allows data to be channelled directly from an I/O device to memory, or vice versa, without any processor intervention. For this reason, DMA is one of the fastest means of passing blocks of data between a peripheral device and memory. Data transfer rates of up to 800 to 900 KB/s are possible on the ISA bus using this technique.

DMA is ideal where large blocks (many kilobytes) of word- or byte-sized data are to be transferred. It is commonly used to implement disk I/O on the PC, but is equally suited to high volume data-acquisition applications.

During a DMA operation, the processor relinquishes control of the system bus to a dedicated DMA controller. Before the data transfer can take place, the DMA controller is programmed with the address of a source or target memory buffer, the number of bytes to be transferred and a number of other parameters. DMA then proceeds under hardware control. The DMA controller manipulates the system bus control lines in order to effect the transfer without involving the processor.

Direct memory access can take place over the ISA bus only in conjunction with a peripheral device that possesses the special circuitry needed to interface to the DMA controller. As we shall see later, all DA&C cards for the PCI bus possess their own bus-control circuitry which lets them initiate bus transfers without the need for a general-purpose DMA controller. A few ISA DA&C adaptor cards provide driver software and/or ROM-based firmware which takes care of programming the DMA controller. In other cases, however, this software may have to be built into the DA&C application itself. The following sections discuss how to program the DMA controller on the PC's ISA bus and give a brief overview of PCI bus mastering.

The DMA controller

All XT bus PCs possess a single Intel 8237A-5 DMA controller. ISA, EISA and MCA machines have either two such controllers

or functionally equivalent custom circuitry. The EISA and MCA controllers provide a high degree of backward compatibility together with a number of useful enhancements, but because these machine-specific features are used in relatively few systems they will not be covered in this section. Readers interested in the enhanced features of the MCA's DMA controller should consult, for example, Eggebrecht (1990), Sanchez and Canton (1994) or van Gilluwe (1994). The latter also describes EISA-specific DMA features. The techniques described in this section can be used for data acquisition on all members of the PC family that possess an ISA, EISA or MCA bus.

DMA channels

DMA controllers provide a number of separate channels for data transfer. The controllers used on the original IBM PC and XT possessed four DMA channels. The additional or enhanced controllers present on ISA, EISA and MCA machines provide a total of eight DMA channels, although some of these channels are dedicated to specific functions and are unavailable for data acquisition.

Table A.1 in Appendix A illustrates the standard DMA channel assignments used in the various classes of PC. In all cases, channels 0 to 3 permit only 8-bit transfers. Channels 5 to 7 (where available) allow data to be transferred 16 bits at a time. These channels do not support 8-bit transfers. Each channel can be programmed to transfer a maximum of 64 K data units. This means that channels 0 to 3 are able to transfer blocks up to 64 KB in length. Because channels 5 to 7 carry words, rather than bytes, data blocks of up to 128 KB can be transferred without having to reprogram the DMA controller.

The dual 8237A arrangement provides a total of seven, rather than eight, usable DMA channels. The first channel of controller 2 (i.e. channel 4) is used for cascading to controller number 1 and is unavailable to application programs.

Channel 0 was used for refreshing the system DRAM on the original IBM PC and so cannot be used for data acquisition. Modern PCs possess dedicated memory refresh circuits, freeing channel 0 for other use. However, the control lines necessary to initiate DMA on channel 0 are not present on the system bus so this channel is also unsuitable for data acquisition. Any of the remaining channels (i.e. 1 to 3 or 5 to 7) can be used for interfacing to DA&C cards provided, of course, that the card supports that channel and that the DMA channel is not already in use.

It is difficult for an application program to determine whether a DMA channel is currently allocated to another device simply by

reading the DMA controller's registers. Although it is possible to discover if a channel is currently in use (i.e. actively transferring data) by monitoring the Status, Address and Count registers (see the section *DMA controller registers* later in this chapter), there is no guarantee that an apparently unused channel will remain so. The responsibility for selecting DMA channels must ultimately rest with the end user.

Types of data transfer

Three types of DMA data transfer operations are possible. The transfer type is programmed by means of bits 2 and 3 of the DMA controller's Mode register. The three transfer types are:

1. Verify
2. Memory to I/O port (also known as DMA read)
3. I/O port to memory (also known as DMA write)

The purpose of the DMA read and DMA write operations should be self-explanatory. The Verify feature performs pseudo-data transfers. It generates DMA cycles with programmed memory addresses, but does not actually read or write data. This mode is not generally used in the PC.

In addition to these transfer modes, it is possible to program the 8237A for memory-to-memory transfers. This type of DMA transfer is also of limited usefulness for a number of reasons. First, it requires channels 0 and 1 to cooperate in the transfer. On the original IBM PC, channel 0 was dedicated to refreshing the system DRAM, making it difficult to use this channel without losing the contents of memory. DRAM refresh is performed by custom circuitry on later PCs. Second, 80386 and subsequent processors can generally perform memory-to-memory transfers more quickly than the DMA controller, by means of their string move (`MOVS` etc.) instructions. Consequently, memory-to-memory DMA is rarely used.

These disadvantages do not apply to DMA read and DMA write operations. Direct memory access is one of the fastest methods of transferring large blocks of data between memory and an I/O port, or vice versa. The remainder of this section will deal only with DMA read and write operations, which are of most relevance to PC-based data acquisition and control.

Overview of the DMA transfer mechanism

Before a DMA transfer can take place, the DMA controller must be programmed with the address of the target (or source) memory buffer, the number of bytes to be transferred, the direction of data

flow and several other parameters which we will discuss later in this section. The software must then enable DMA on the selected channel. After the controller has been properly configured, the adaptor card initiates the transfer process (possibly in response to a hardware event or as a result of a command issued by the software). The transfer proceeds as follows.

1. Whenever an adaptor card wishes to perform DMA it asserts the appropriate DMA Request line. The DMA controller possesses one DMA Request line for each channel. The XT bus makes three of these request lines, DREQ1, DREQ2 and DREQ3, available to adaptor cards. The ISA bus provides an additional three DMA Request lines: DREQ5, DREQ6 and DREQ7. The remaining request lines, DREQ0 and DREQ4, are used internally and are not available on the expansion bus.
2. When the DMA controller senses the DREQn signal it first checks to ensure that DMA is enabled for that channel (i.e. the channel denoted by n). DMA channels can be individually enabled and disabled by software. The controller also prioritizes DMA requests with any that may be pending on other channels.
3. If DMA is enabled, the DMA controller asserts its Hold Request (HRQ) line. The processor responds to this signal when the bus becomes idle by freeing the system bus and issuing a Hold Acknowledge (HLDA) signal to the 8237A DMA controller. This, in turn, asserts the Address Enable (AEN) line and places the address of the source or target memory location onto the address bus. This is shortly followed by activation of the appropriate DMA Acknowledge (DACKn) line (each DMA channel has its own DACK line).
4. The adaptor card detects the DACKn signal which informs it that the data transfer is now in progress.
5. The DMA controller, having taken over the system bus, asserts the appropriate I/O or memory read/write lines. In the case of a DMA Write operation, the $\overline{\text{IOR}}$ and $\overline{\text{MEMW}}$ lines are asserted. For a DMA Read, the $\overline{\text{MEMR}}$ and $\overline{\text{IOW}}$ lines are asserted. This causes data to be transferred directly between the I/O device and memory. The DMA controller adjusts the target (or source) memory address after each transfer has been completed so that subsequent transfers access the next byte or word in the memory buffer.
6. Depending upon the transfer mode selected, the adaptor card may release the DREQn line after each byte or word has been transferred or at other times necessary to regulate the flow of data. In response, the DMA controller releases the HRQ line enabling the processor to take control of the bus. The whole

process repeats until the specified number of bytes or words have been transferred.

7. When the programmed number of bytes or words have been transferred, the DMA controller asserts the Terminal Count (TC) line of the system bus. This informs the adaptor card that the transfer operation is complete. The DMA controller may then either automatically disable DMA on the current channel or, if autoinitialization has been selected (see the following section), prepare itself for another DMA sequence.

You may be wondering how the adaptor card's I/O port is selected, if the address bus holds only a memory address. It is, in fact, the receipt of the DACKn signal, rather than decoding of an I/O address, that enables the contents of the I/O port onto the data bus. Other I/O ports, which may otherwise decode the memory address, are prevented from doing so by the AEN signal issued by the DMA controller. The AEN line is asserted only when a DMA bus cycle is in progress. This signal is used on the system bus to disable normal I/O port address decodes. The DMA process is summarized in Figure 6.1. The circled letters denote the order in which the various operations take place.

A more detailed account of the transfer procedure is provided by Eggebrecht (1990). Most of the handshaking that occurs during DMA is transparent to the programmer. It is only necessary to understand that the adaptor card initiates, and in some cases regulates,

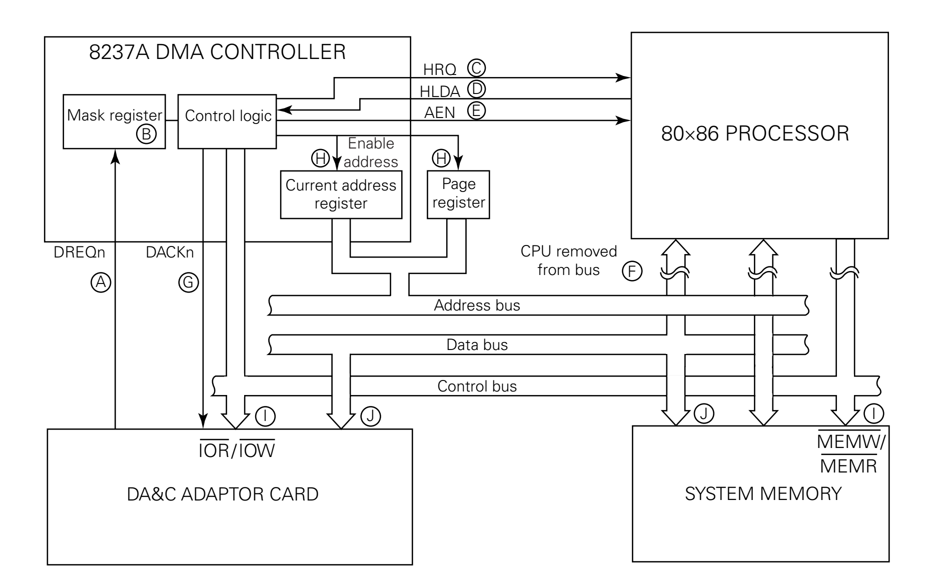


Figure 6.1 *Schematic illustration of the DMA process*

data transfer by means of a selected DREQn line. The DREQn line is used in a variety of ways, depending upon the programmed transfer mode (see *DMA transfer modes* later in this chapter) to control the flow of data and interweaving of DMA and processor bus cycles.

Autoinitialization

The 8237A DMA controller possesses a number of 16-bit registers for each channel. Two of these hold the current memory address for the transfer and the current word count (i.e. the number of bytes or words transferred). These values are incremented or decremented, as appropriate, on each transfer cycle. When the 8237A is first programmed, the initial memory address and word count are loaded into these registers. The initial values are also recorded in two other registers, the Base Address and Base Word Count registers. The values held in these registers do not change during the DMA process.

The 8237A can be programmed (via the Mode register) to automatically reinitialize the Current Address and Current Word Count registers at the end of a programmed DMA sequence. During this autoinitialization, the contents of the Base Address and Base Word Count registers are copied to the associated Current Address and Current Word Count registers, thereby preparing the 8237A for another DMA sequence. The DMA channel remains enabled so that the DMA sequence can be repeated as soon as the next DREQ signal is detected. If the autoinitialization facility is not enabled, the DMA channel disables itself (by setting the appropriate Mask bit) after the programmed quantity of data has been transferred.

DMA priorities

Although the 8237A can be programmed to operate according to one of two priority schemes – fixed or rotating – the PC should generally only operate the 8237A in the fixed priority mode. In this mode, channel 0 (memory refresh) always has the highest priority, channel 1 the next highest and so on. The dual-controller arrangement employed on ISA, EISA and MCA systems extends the priority scheme to the second controller. Thus the priority order is channel 0, 1, 2, 3, 5, 6 and 7 (remember that channel 4 is used for cascading the two controllers and is not available for interfacing to peripheral devices). If one or more devices request DMA service while a transfer is in progress on another channel, they must wait until the current transfer is complete. The device with the highest priority will then be serviced first.

DMA transfer modes

Apart from a special Cascade mode which is used for connecting dual DMA controllers, the 8237A provides three data transfer modes. These can be selected via the controller's Mode register (see *DMA controller registers* later in this chapter). Note that the adaptor card hardware must be specifically designed to operate in each mode. You should use only those modes that are supported by your hardware.

Single Transfer mode

In this mode only one byte or word is transferred at a time and when each transfer is complete, the 8237A releases the system bus to the processor. If the adaptor card holds DREQn active throughout the transfer, the processor will be allowed only one bus cycle before the 8237A reasserts the HRQ line and takes control once more. In this way ordinary processor bus cycles can be interwoven with DMA cycles.

Demand Transfer mode

This mode allows the adaptor card to regulate the DMA transfer by temporarily deactivating DREQn. While DREQn is active the transfer proceeds in much the same way as the Single Transfer mode except that no processor bus cycles are interwoven with the DMA cycles. The controller will continue with the transfers (provided DREQn remains active) until the programmed number of bytes or words has been transferred.

Block Transfer mode

In Block Transfer mode, the device issues one DREQn pulse to initiate the transfer of a whole data block (i.e. the number or bytes or words specified in the Base Word Count register). Processor bus cycles are not interwoven with the DMA cycles. The DREQn signal need not be asserted throughout the transfer; it may go inactive as soon as the DACKn signal becomes active.

DMA controller registers

Each DMA controller is programmed via a number of internal registers. These are listed in Table 6.4. The first controller (which supplies DMA channels 0 to 3) is located at I/O port base address 0000h. The second 8237A in dual-controller systems has a base address of 000Ch. Note that writes to addresses 0Ch, ODh, 0Eh, D8h, DAh and DCh do not directly access any registers. The actual value of the data written to these addresses is unimportant, however.

Table 6.4 *8237A DMA controller register map*

Port	*Direction*	*Controller*	*Description*
00h	R/W	1	Channel 0: Current/Base Address.
01h	R/W	1	Channel 0: Current/Base Word Count.
02h	R/W	1	Channel 1: Current/Base Address.
03h	R/W	1	Channel 1: Current/Base Word Count.
04h	R/W	1	Channel 2: Current/Base Address.
05h	R/W	1	Channel 2: Current/Base Word Count.
06h	R/W	1	Channel 3: Current/Base Address.
07h	R/W	1	Channel 3: Current/Base Word Count.
08h	R	1	Status register.
08h	W	1	Command register.
09h	W	1	Request register.
0Ah	W	1	Mask register.
0Bh	W	1	Mode register.
0Ch	W	1	Not a register. Writing to this address clears the byte pointer flip-flop.
0Dh	R	1	Temporary register.
0Dh	W	1	Not a register. Writing to this address resets the controller.
0Eh	W	1	Not a register. Writing to this address clears the Mask register.
0Fh	W	1	Write-all-mask register.
C0h	R/W	2	Channel 4: Current/Base Address.
C2h	R/W	2	Channel 4: Current/Base Word Count.
C4h	R/W	2	Channel 5: Current/Base Address.
C6h	R/W	2	Channel 5: Current/Base Word Count.
C8h	R/W	2	Channel 6: Current/Base Address.
CAh	R/W	2	Channel 6: Current/Base Word Count.
CCh	R/W	2	Channel 7: Current/Base Address.
CEh	R/W	2	Channel 7: Current/Base Word Count.
D0h	R	2	Status register.
D0h	W	2	Command register.
D2h	W	2	Request register.
D4h	W	2	Mask register.
D6h	W	2	Mode register.
D8h	W	2	Not a register. Writing to this address clears the byte pointer flip-flop.
DAh	R	2	Temporary register.
DAh	W	2	Not a register. Writing to this address resets the controller.
DCh	W	2	Not a register. Writing to this address clears the Mask register.
DEh	W	2	Write-all-mask register.

Simply performing an OUT instruction to these addresses (with any data) initiates the actions listed in the table.

In addition to the registers present within the 8237A itself, all members of the PC family possess a set of page registers that are used in DMA memory addressing. These are not contained in the 8237A itself. Instead 74LS612 Memory Mapper ICs, or equivalent devices, supply the necessary registers. Page registers are required because the 8237A's internal address registers are 16 bits wide and so can address only 65 536 different memory locations. In order to access any region of the PC's memory, the page registers are programmed with the most significant bits of the physical memory address for each transfer, as indicated in Table 6.5. On XT-bus systems, only the lower 4 bits of the page register are required for accessing any part of available memory (i.e. up to 1 MB). The low order nibble of the page register contains address bits A_{16} to A_{19}. Bits A_0 to A_{15} are programmed into the 8237A itself.

ISA, EISA and MCA systems use either 7 or 8 bits of each page register in order to access physical addresses within the first 16 MB. In the case of channels 0 to 3, the 8237A is programmed with address bits A_0 to A_{15} and the page register contains bits A_{16} to A_{23} as shown in Figure 6.2. In order to access 16-bit words at even memory addresses, address bit A_0 is ignored on channels 5, 6 and 7. For these channels, the 8237A is programmed with address bits A_1 to A_{16} while the page register holds bits A_{17} to A_{23}.

Because of the need to use page registers, the location and size of memory buffers is restricted. Transfers on channels 0 to 3 must not cross an absolute 64 KB address boundary and consequently may not exceed 64 KB in total. Similarly, 16-bit transfers on channels 5 to 7 must not cross a 128 KB boundary and so cannot exceed 128 KB. Transfers that cross these address boundaries require the

Table 6.5 *Page register map*

	PC and XT		*AT, MCA and EISA*	
I/O port	*DMA channel*	*Address lines*	*DMA channel*	*Address lines*
81h	2	A16–A19	2	A16–A23
82h	3	A16–A19	3	A16–A23
83h	1	A16–A19	1	A16–A23
87h			0	A16–A23
89h			6	A17–A23
8Ah			7	A17–A23
8Bh			5	A17–A23

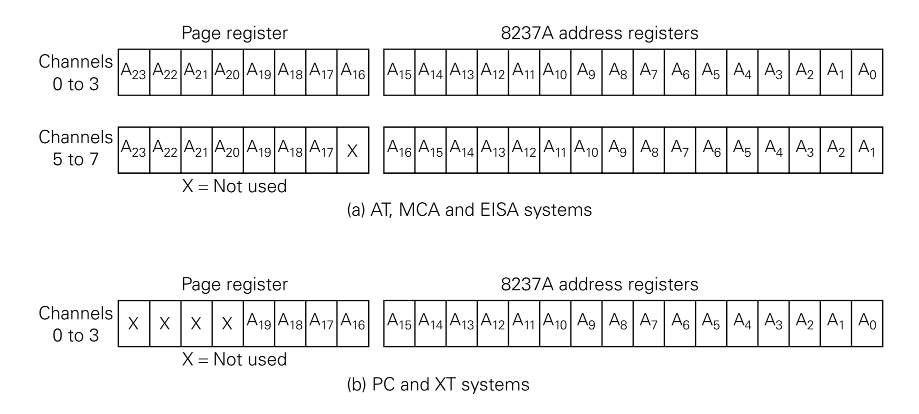


Figure 6.2 *Address mapping using page registers*

controller's address register and associated page register to be reinitialized by software. Some embedded systems avoid these problems by employing DMA controllers with a larger addressing capability. These are, unfortunately, unavailable on standard ISA PCs.

A note on channel numbers

The following sections describe the registers present in a single 8237A DMA controller. Because ISA, EISA and MCA systems possess two such controllers (or compatible custom circuits), the same information also applies to the DMA channels of the second controller. Channel numbers 0, 1, 2 or 3 referred to in the following discussion represent either channels 0 to 3 in the case of the first controller, or channels 4 to 7 in the case of the second controller.

Current Address and Base Address registers

Each channel has a Current Address and Base Address register. These 16-bit registers are initialized together in one operation by software. The address is written in two 8-bit bytes. The low byte is written first and this must always be followed by the high byte. The contents of the Current Address register are either incremented or decremented when each byte or word is transferred (increment/decrement is software selectable). Reading from these register addresses returns the value of the Current Address register. The Base Address register, which is used to implement the autoinitialization function, always retains the last value written.

The 2-byte read and write operations are controlled by an internal Byte Pointer flip-flop. This is toggled each time a byte is read or written. When the flip-flop is clear, the controller receives or

supplies the low order byte of the address. When it is set, the controller processes the high order byte. It is wise to clear the flip-flop before commencing any read or write operations. This may be accomplished by writing any value to I/O port 0Ch (or D8h in the case of the second controller).

Current Word Count and Base Word Count registers

Each channel also has a Current Word Count and Base Word Count register. The word count is written in two 8-bit bytes. The low order byte is written first and this must always be followed by the high byte. The contents of the Current Word Count register are decremented after each byte or word is transferred. When the count reaches zero the next transfer causes the count to roll over to FFFFh which signifies the end of the transfer. The Current Word Count register always holds the number of transfers to be performed, minus 1. If, for example, 0800h bytes are to be transferred, you should initialize the Current Word Count register with the value 07FFh.

Reading from these I/O addresses returns the value of the Current Word Count register. The Base Word Count register, which is used to implement the autoinitialization function, always retains the last value written.

The two-byte read and write operations are controlled by an internal Byte Pointer flip-flop. This is toggled each time a byte is read or written. When the flip-flop is clear, the controller receives or supplies the low order byte of the word count. When it is set, the controller processes the high order byte. It is wise to clear the flip-flop before commencing any read or write operations. This may be accomplished by writing any value to I/O port 0Ch (or D8h in the case of the second controller in AT systems).

Status register

The Status register is a read-only port that provides the application program with information about the current state of the DMA controller. Bits 0 to 3 are set when the corresponding channel has reached its terminal count (i.e. when the programmed number of bytes or words has been transferred). These bits are automatically cleared after the Status register has been read. Bits 4 to 7 are set high whenever a DREQ is active on DMA channels 0 to 3. This is summarized in Table 6.6.

Command register

To maintain hardware compatibility, most of the bits in this write-only register should be zero on the PC. Only bit 2 is normally manipulated

Table 6.6 *The Status register (read only)*

Bit	Controller 1	Controller 2
0	1 = Channel 0 terminal count	Used for channel 4 cascade
1	1 = Channel 1 terminal count	1 = Channel 5 terminal count
2	1 = Channel 2 terminal count	1 = Channel 6 terminal count
3	1 = Channel 3 terminal count	1 = Channel 7 terminal count
4	1 = DREQ0 pending	Used for channel 4 cascade
5	1 = DREQ1 pending	1 = DREQ5 pending
6	1 = DREQ2 pending	1 = DREQ6 pending
7	1 = DREQ3 pending	1 = DREQ7 pending

by PC software. This bit enables or disables the controller and is used to prevent the controller from responding to DREQ signals while it is being programmed. Setting bit 2 disables the controller, and clearing the bit enables the controller. Note that, in order to avoid disrupting the memory refresh subsystem, you should not disable the DMA controller in XT-bus machines. For information on the remaining bits in this register you should consult the Intel 8237A-5 data sheet.

Request register

The Request register allows DMA requests to be initiated by software rather than by a hardware DREQ signal. The binary-coded channel number is loaded into bits 0 and 1 (channels 4 to 7 on the second DMA controller should be coded as 00b to 11b respectively). Bit 2 controls the setting of the controller's internal DREQ signal. This bit should be set in order to perform a software DMA request. So, to initiate a DMA request on channel 1, for example, you should write the value 00000101b to the request register. Table 6.7 summarizes the operation of the Request register.

Table 6.7 *The Request register (write only)*

Bit	Description
1,0	Channel number to which the request applies (channels 4–7 of controller 2 are represented by bit patterns 00b to 11b).
2	0 = Clear request. 1 = Initiate DMA request.
7–3	Not used.

Table 6.8 *The Mask register (write only)*

Bit	*Description*
1,0	Channel number to which the mask bit applies (channels 4–7 of controller 2 are represented by bit patterns 00b to 11b).
2	0 = Enable DMA channel. 1 = Disable DMA channel.
7–3	Not used.

Mask register

This is a write-only register. It is used for selectively enabling or disabling DMA channels according to the scheme shown in Table 6.8. A hardware or software reset will set all mask bits, disabling all DMA channels. Only those channels actually used should be enabled. You should not disable channel 0 on systems that use it for refreshing memory.

Mode register

The Mode register determines how the 8237A operates. It controls the type of transfer, autoinitialization, address increment/decrement selection and the transfer mode to be used. The bit assignments in this write-only register are listed in Table 6.9.

Temporary register

This read-only register holds data between read and write cycles during memory-to-memory transfers. It is of little interest for data acquisition.

Write-All-Mask register

This allows DMA channels to be enabled or disabled in one operation. The normal Mask register permits control only of individual channels. The bit assignments for the Write-All-Mask register are shown in Table 6.10. This is a write-only register. Alternatively, if it is necessary to enable all four DMA channels, your software can simply write any value to address 0Eh (for controller 1) or DCh (for controller 2). Only those channels actually used should be enabled. You should not disable channel 0 on systems that use it for refreshing memory.

DMA in protected and V86 modes

During DMA transfers the address contained in the 8237A's Current Address and Page registers refers to physical memory. This causes

Table 6.9 *The Mode register (write only)*

Bit	*Description*
1,0	Channel number to which the mode settings apply (channels 4–7 of controller 2 are represented by bit patterns 00b to 11b).
3,2	Transfer type (ignored in cascade mode): 00b = Verify 01b = DMA write (I/O to memory) 10b = DMA read (memory to I/O) 11b = Illegal.
4	0 = Disable autoinitialization. 1 = Enable autoinitialization.
5	0 = Increment address during DMA. 1 = Decrement address during DMA.
7,6	Transfer mode: 00b = Demand mode 01b = Single mode 10b = Block mode 11b = Cascade mode.

Table 6.10 *The Write-All-Mask register (write only)*

Bit	*Controller 1*	*Controller 2*
0	Channel 0 mask. 0 = Enabled	Channel 4 mask. Should be 1 on PC
1	Channel 1 mask. 0 = Enabled	Channel 5 mask. 0 = Enabled
2	Channel 2 mask. 0 = Enabled	Channel 6 mask. 0 = Enabled
3	Channel 3 mask. 0 = Enabled	Channel 7 mask. 0 = Enabled
7–4	Not used	Not used

problems with software running in the protected and virtual 8086 modes offered by 80386 and later processors. Because of the selector addressing and page translation mechanisms used in these modes, the application software that is responsible for programming the DMA controller has no knowledge of the physical memory address of its DMA buffer.

Some memory managers address this problem by using the processor's I/O protection mechanisms (see Chapter 1) to trap accesses to the DMA controller. The memory manager can then translate the address of the application program's virtual buffer into a physical address. A temporary *mirror buffer* may be allocated by the memory manager if the physical address falls outside the

16 MB addressable range of the DMA controller. This intermediate buffering stage may significantly affect the throughput of DA&C application. However, provided that DMA buffers are allocated within the 16 MB range, this technique should not affect the real-time performance of the system.

Microsoft Windows virtualizes DMA by providing a set of Virtual DMA software services. These services are essential in the '386 Enhanced Mode of Windows 3.1 and in later versions of Windows or when independent bus master DMA controllers are used. Bus masters are additional DMA controllers that may be provided as an integral part of an I/O device. Because the I/O addresses of the bus master's registers are not fixed (as they are with the PC's standard DMA controllers) it is more difficult for the operating system or memory manager to trap I/O accesses to their registers. For this reason, the DA&C application should not attempt to access the DMA controller directly. Instead, all DMA requests must be routed via the operating system's Virtual DMA services. These include function calls for allocating DMA buffers, for copying data to and from the DMA buffers, and for locking memory addresses in order to prevent remapping or conflicts with other DMA operations. As with the virtual I/O system used under Microsoft Windows, the overhead incurred with virtual DMA can seriously affect overall data-acquisition rates, especially in high speed applications. Further information on virtual DMA may be found in the texts by Brown and Kyle (1991) and van Gilluwe (1994).

DMA programming

Programming a system for DMA involves configuring two components: the peripheral DA&C device which supplies or receives data, and the DMA controller itself. The DA&C device is usually configured via one or more control registers. Because of the wide variety of data-acquisition cards available, we will not discuss the DMA facilities offered by individual devices. You should consult your DA&C interface card manual for programming details.

Instead, this section illustrates how the PC's DMA controller can be programmed to manage the I/O transfer. After programming the 8237A controller, data transfer is usually initiated in one of three ways:

1. Software commands issued direct to the DA&C device, causing it to activate DREQ.
2. Software commands issued to the 8237A's Request register.
3. Hardware signals such as event triggers or periodic clock pulses.

Programming the DMA controller is quite straightforward provided that you take a few fairly simple precautions. Most of these are just common sense, but are listed here as they can be easily overlooked.

- Your software should ensure that DMA requests are disabled on the channel that is being programmed. This will prevent the controller from attempting to service a DMA request until the buffer addresses and word counts etc. have been properly configured. Only enable the DMA channel after programming is complete.
- It is also a sensible precaution to disable interrupts in order to prevent other processes from accessing the 8237A until it has been fully programmed.
- Only enable those channels that you actually use, and do not alter the mask bits of any other channels.
- Before terminating your program or disposing of a memory buffer, always ensure that the DMA channel is left disabled.
- Before writing address and word count values, clear the Byte Pointer flip-flop by outputting any value to I/O port 0Ch (for channels 0 to 3) or D8h (for channels 5 to 7).
- When loading the Address and Page registers (particularly for channels 5 to 7), be sure to preserve the bit pattern indicated in Figure 6.2.
- Load the Count registers with a value one less than the number of bytes (or words in the case of channels 5 to 7) to be transferred.
- Avoid using Block Transfer mode, particularly on XT class machines, where this mode might interfere with the memory refresh subsystem.
- Use the smallest memory buffers consistent with your application.

Listing 6.1 illustrates how a DMA channel can be configured. For the sake of clarity, the various DMA parameters and register addresses are passed to the `SetupDMA` procedure in the form of global variables. In a real program, all of these variables would have to be initialized before calling `SetupDMA`. Separate code and data segments are not shown in the listing. However, the code assumes that DS has been initialized to point to the data segment. The `SetupDMA` routine itself should be self explanatory.

Data acquisition using DMA

DMA is an essential technique for high speed data acquisition. It is suitable for collecting ADC data as it is digitized; for reading the contents of on-board memory buffers or for transferring data to and from a communications interface card such as an IEEE-488 adaptor. It is also an ideal mechanism for signal generation. Data can be

Listing 6.1 *Configuring 8237A channel 7 for a DMA write operation*

```
;Register addresses
PageRegAddr   dw      ;Address of page register
AddrRegAddr   dw      ;Address of address register
CountRegAddr  dw      ;Address of count register
MaskAddr      dw      ;Address of Mask register
ModeAddr      dw      ;Address of Mode register
FlipFlopAddr  dw      ;Address of Clear Flip Flop port

;Variables for SetupDMA
Controller    db      ;Controller number (1 or 2)
Channel       db      ;8237A channel number (0 to 3)
BufOfs        dw      ;Pointer to buffer (buffer must not cross an
BufSeg        dw      ;  absolute 64K / 128K boundary).
Count         dw      ;Number of bytes/words to be transferred
Direction     db      ;0 = Output (DMA read); 1 = Input (DMA Write)
Mode          db      ;0 = Demand; 1 = Single; 2 = Block

               |             |
               |             |

SetupDMA      PROC  FAR
              ;Sets up a DMA channel according to the parameters listed above.
              ;Address increment (rather than decrement) is always selected and
              ;autoinitialization is always turned off.
              ;Entry: Controller, Channel, BufOfs, BufSeg, Count, Direction
              ;         and Mode variables, as well as the various register
              ;         addresses, must all be defined.
              ;       DS must point to the segment containing these variables.
              ;       Other registers may contain any values.
              ;Exit:  AX, BX, CX, DX and Flags registers are corrupted.

              ;Convert BufSeg:BufOfs into 24-bit physical address in BL,CX.
              mov    ax,BufSeg          ;AX = Segment of buffer
              xor    bx,bx              ;BX = 0
              mov    cx,4               ;Loop counter
              clc                       ;Clear Carry Flag
Multiply16:   rcl    ax,1               ;Rotate BX,AX left via Carry Flag
              rcl    bx,1               ;
              loop   Multiply16         ;Repeat 4 times to multiply BX,AX by 16
              add    ax,BufOfs          ;Add buffer offset
              adc    bx,0               ;Add Carry Flag in case of carry from ADD
              mov    cx,ax              ;BX,CX now holds the physical adddress

              ;Check controller
              mov    al,Controller      ;Get DMA controller number
              cmp    al,2               ;Is it controller 2 ?
              je     Ctrl2              ;  Yes, adjust count and address
              push   Count              ;  No, no need to adjust
              jmp    LoadRegs           ;

Ctrl2:        ;Controller 2, so adjust count and address for word transfer
              rcr    bl,1               ;CF = A16; MSB of BL is undefined
              rcr    cx,1               ;A16 --> MSB of CX; CF = A0
              rcl    bl,1               ;Restore page reg bit pattern; LSB = A0
              mov    ax,Count           ;Get number of bytes
              shr    ax,1               ;AX is now number of words
              push   ax                 ;Save on stack
```

Listing 6.1 *(continued)*

```
LoadRegs:     cli                        ;Disable interrupts

              ;Mask (disable) DMA channel
              mov    dx,MaskAddr         ;Address Mask register
              mov    al,Channel          ;Channel number
              or     al,04h              ;Set mask bit
              out    dx,al               ;Load Mask register

              ;Load page register
              mov    dx,PageRegAddr      ;Address Page register
              mov    al,bl               ;Get high order address bits
              out    dx,al               ;Load Page register

              ;Clear Byte Pointer flip flop
              mov    dx,FlipFlopAddr     ;Address Flip Flop Control
              out    dx,al               ;Clear flip flop

              ;Write 8237A address register
              mov    dx,AddrRegAddr      ;Address 8237A's Address register
              mov    al,cl               ;Load low byte
              out    dx,al               ;
              mov    al,ch               ;Load high byte
              out    dx,al               ;

              ;Write Count register
              mov    dx,CountRegAddr     ;Address Count register
              pop    ax                  ;Get byte/word count from stack
              dec    ax                  ;Count is one less than no. of transfers
              out    dx,al               ;Output low byte
              mov    al,ah               ;  followed by
              out    dx,al               ;  high byte.

              ;Write Mode register
              mov    dx,ModeAddr         ;Address Mode register
              mov    al,Channel          ;Channel number
              mov    ah,Direction        ;Include Direction bits
              mov    cx,2                ;
              shl    ah,cl               ;
              or     al,ah               ;
              mov    ah,Mode             ;Include Mode bits
              mov    cx,6                ;
              shl    ah,cl               ;
              or     al,ah               ;
              out    dx,al               ;Load Mode register

              ;Unmask (enable) DMA channel
              mov    dx,MaskAddr         ;Address Mask register
              mov    al,Channel          ;Define channel. Mask bit is left clear
              out    dx,al               ;Load Mask register

              sti                        ;Enable interrupts

              retf                       ;Return to caller

SetupDMA      ENDP
```

easily clocked out from the PC's memory to a device controlled by a hardware pacer clock. Both DMA read and write operations can be performed in the background with minimal disturbance to the foreground DA&C program.

DMA transfer rate

The maximum theoretical DMA transfer rate, which would be achievable only in Block Transfer mode, can be calculated by multiplying the number of bus clocks required to transfer each byte by the length of each clock period.

On the XT-bus systems, each DMA read/write transfer takes at least six bus clocks. A higher number of clock intervals are required if bus wait states are used. Bus frequencies of 4.77, 8 and 10 MHz are commonly used on XT compatible systems, although in 8 and 10 MHz systems the DMA controller may operate at one half of the bus clock frequency. A 4.77 MHz XT system will take approximately 1260 ns to transfer 1 byte.

ISA systems require at least five bus clocks to transfer each byte or word. A 10 MHz ISA PC may therefore take 500 ns to perform a single transfer, so the maximum theoretical transfer rate is about 2 MB/s. These figures will, of course, vary with bus clock speed.

The maximum transfer rate is rarely achieved, however. Delays due, for example, to the finite ADC conversion time and multiplexer settling time may restrict throughput. The DMA controller is also usually programmed to operate in Single Transfer (or occasionally Demand Transfer) mode. This allows normal processor bus cycles to be interwoven with DMA cycles and consequently limits the maximum achievable transfer rate. Fast ADC cards that provide DMA facilities will typically provide sustained throughputs of the order of 50 000–250 000 samples/s (i.e. about 100–500 KB/s). However, some high speed cards are claimed to allow burst DMA rates approaching 2 MB/s over a 10 MHz ISA bus.

Dual-channel DMA

The limited DMA buffer size of 64 KB (or 128 KB for channels 5 to 7) can be a serious drawback. In order to stream a larger quantity of data to the PC's memory, it is necessary to suspend data acquisition whenever the terminal count is reached so that the DMA controller can be reprogrammed with the address of a new buffer. The DA&C system may be unable to sample data during this time and there is a danger that important readings will be lost. Average throughput rates can be significantly reduced if more than 64 KB (or 128 KB for channels 5 to 7) are to be transferred. This is a particularly severe problem in high speed applications.

One solution is to employ dual-channel DMA. This requires special hardware support, but is relatively straightforward to implement. Two DMA buffers are allocated and a separate DMA channel is set up for each buffer. The digitized readings are transferred via one DMA channel and when this reaches its terminal count the DA&C adaptor card switches to the second channel. The terminal count signal also causes the card to issue a hardware interrupt. The software can respond to the interrupt either by reading and processing the first buffer or by reconfiguring the first DMA channel so that it addresses a third buffer. The procedure is repeated when the second channel reaches its terminal count, allowing data to be transferred alternately via the two DMA channels.

Dual-channel DMA is most useful when data is transferred in short isolated bursts. This allows the processor sufficient time between bursts to respond to the terminal count interrupt and to perform any other processing that may be necessary. DA&C cards which support dual-channel DMA also often incorporate FIFO memory buffers. These are usually large enough to hold 1–2 KB of data (sometimes considerably more). When sufficient data has been recorded in the buffer, it is transferred in small blocks (typically 256 or 512 bytes) using the dual-channel DMA technique.

DMA latency

It is not only the data transfer rate which may be important in a DA&C application. The time between assertion of the DREQ line and transferring the first data byte is often an equally crucial consideration. This latency time depends upon the priority of the DMA channel and whether other DMA requests are pending. The minimum time for completion of a single-byte transfer (i.e. a full DMA write or read cycle) is at least six bus clocks on the XT bus or five clocks on ISA and MCA machines. Additional clock cycles will be required if the system is configured to include bus wait states. The latency time will typically be longer than this minimum transfer time. If a DMA channel is programmed for multiple-byte transfers this can increase the latency of other channels.

When should you use DMA?

Although DMA is one of the fastest methods for transferring large quantities of data, it is not always the most appropriate technique. You should consider the following points when deciding whether to use DMA.

- Would programmed I/O be fast enough? For relatively low acquisition rates, you may prefer the simplicity of polled or

interrupt-driven I/O. The throughput obtainable with these techniques will be highly dependent upon the speed of the DA&C hardware as well as on the amount of processing to be performed by the software. Assembly language routines may achieve rates of up to about 20 000–30 000 samples/s without the benefit of hardware buffering (i.e. direct from an ADC). Higher acquisition rates may be possible by using a tight polling loop.

- Will DMA provide an adequate throughput? Most DA&C hardware manufacturers provide typical DMA throughput figures. If you require a higher throughput than is possible using DMA, or if the DMA latency is unacceptable, it may be necessary to use a DA&C card that provides high speed buffered input. Burst acquisition rates of up to a few MHz are supported by some devices of this type. At the end of a data-acquisition run, the contents of the card's memory buffer can be transferred to the PC's memory (albeit somewhat more slowly) by using either programmed input or DMA techniques. The rate at which this transfer is performed is usually also an important consideration.
- Would programmed I/O be faster than DMA? Single or Demand Transfer DMA can be used for reading data from hardware buffers. These techniques provide transfer rates from several hundred KB/s up to approximately 1 MB/s. On 80286 and later processors the `REP INSW` instruction allows data to be transferred from a hardware buffer at up to about 1 to 2 MB/s, depending upon processor type. This is significantly faster than DMA. The 32-bit `REP INSD` instruction may provide an additional increase in throughput, but because of delays inherent in the DA&C hardware, 32-bit I/O will not generally provide twice the throughput of 16-bit transfers. Whether 16-bit or 32-bit transfers are employed, the hardware registers must, of course, be capable of responding to back-to-back I/O instructions. `REP INSW` and `REP INSD` are only suitable for reading buffered data. ADCs cannot generally supply a sequence of digitized readings quickly enough to satisfy the repeated input requests.
- Will DMA programming overheads be significant? You should consider whether the overhead involved in reprogramming the DMA controller will exceed the time saved by using DMA. This will, of course, depend upon the DMA rate achievable and the speed of the processor. It will be relatively more efficient to use programmed I/O with faster processors. This consideration is only relevant if the 8237A programming is carried out in a time-critical portion of the program.
- How will DMA bus cycles affect the software? Interweaving of bus cycles in Single Transfer mode will reduce the average execution

speed of the DA&C program by approximately one half. Because the DMA controller takes over the system bus whenever it needs to service a DREQ, DMA cycles take precedence over even high priority interrupt handlers and tasks. Systems that use Demand Transfer mode will also periodically suspend processing while blocks of data are transferred.

- Is the data stream suitable for DMA? DMA is intended for transferring a regular stream of data to or from the PC's memory. If individual readings, or blocks of varying size, are to be input at irregular intervals, it might be more appropriate to use polled or interrupt-driven I/O.
- Will background operation be important? DMA is particularly suited to background data acquisition. Once the DA&C hardware and DMA controller are configured, data acquisition can proceed with very little software intervention.
- Are there other reasons to avoid polled or interrupt-driven I/O? Data-acquisition programs running under non-deterministic operating systems and/or those with high interrupt latencies, such as Microsoft Windows, may benefit from the more predictable response of DMA-based hardware techniques.

PCI bus mastering

The preceding discussion relates to the DMA system available on the ISA, EISA and MCA buses. Transfers analogous to DMA can also take place on the PCI bus, although a somewhat different and more flexible approach is adopted. The PC's motherboard does not provide a general-purpose DMA controller for the PCI bus. Instead the system allows for *bus mastering*. Each PCI device (e.g. adaptor card) possesses its own special DMA-type circuitry for initiating control of the PCI bus. This allows any PCI device to communicate with another without involving the processor. A DA&C card could, for example, continuously acquire data at a high rate into an on-board FIFO buffer and periodically transfer the buffer contents over the PCI bus into system memory. The whole process can be carried out without processor intervention, other than that required to initially program the DA&C card and, perhaps, trigger the acquisition sequence. This capability provides a means for high speed data transfers that have a minimal effect on software execution times. 32-bit implementations of the PCI bus, clocked at 33 MHz, can transfer data to or from a contiguous block of memory at up to 132 MB/s. This requires that a special addressing mode (burst mode) is used. The maximum data rate drops to 44 MB/s for normally addressed data (multiplexed mode).

The PCI bus arbitrates between different devices wishing to take control of the bus. To request control of the bus, a bus master (on, for example, a DA&C card) will activate the $\overline{\text{REQ}}$ bus line. The PCI arbitration logic then asserts the $\overline{\text{GNT}}$ line, passing control of the bus to the requesting device (which is known as the initiator).

The transfer is similar in principle to ISA-based DMA, although there are some important differences. The initiator provides the 32-bit (or 64-bit) address of the target device, placing it on the bus's Address/Data lines. Addressing is performed in one of two ways. In burst mode the target address for the first transfer is transmitted over the bus and then the target device calculates the address for each subsequent transfer by incrementing the address by the data size (4 or 8 bytes). As the bus undergoes only an initial addressing phase, transfer speed is maximized, but it is possible to access only contiguous blocks of memory in this way. In multiplexed mode, however, each transfer is explicitly addressed. It is these additional addressing phases that reduce bus throughput. The type of data transfer – e.g. memory read, memory write, I/O read or I/O write – is specified by sending a command (i.e. a bit pattern on special bus lines) to the PCI bus logic.

The initiator indicates the start of a transfer by asserting the $\overline{\text{FRAME}}$ bus line. The initiator and target then control the transfer sequence via the $\overline{\text{IRDY}}$ and $\overline{\text{TRDY}}$ lines. When the transfer is complete, the initiator deactivates the $\overline{\text{FRAME}}$ signal (see Buchanan (1999)).

An important feature of the PCI bus mastering system is that it allows DA&C cards with a degree of on-board intelligence to independently initiate and control the transfer of large quantities of digitized data into system RAM. Some DA&C hardware manufacturers, such as National Instruments, have developed optimized PCI bus master circuits which employ techniques analogous to dual-channel DMA. These facilitate continuous high speed transmission of acquired data into multiple buffers or non-contiguous memory blocks.

6.3 Buffers and buffered I/O

As we have seen in the previous section, buffering is a useful technique for decoupling DA&C hardware interfaces from the supervising software. By providing temporary storage for acquired data it is possible to average out the irregularities in software timing that are introduced by interrupt latencies, task switching or DMA operations. This allows data acquisition to proceed at a regular and guaranteed rate. Memory buffers are normally used in conjunction

with DMA and interrupt-driven data-acquisition systems to facilitate asynchronous I/O. Choosing the correct type of buffering system can greatly simplify subsequent management of data. We will consider two classes of buffer: hardware memory buffers, which are managed by the data-acquisition device, and software buffers maintained by the DA&C application program itself.

Hardware buffering techniques

Many DA&C devices have a limited capacity for on-board buffering of acquired data. FIFO buffers ranging from typically 1 to 64 KB are used on some of the more sophisticated dumb data-acquisition cards. Intelligent devices are often equipped with considerably larger data buffers.

Acquired data can be channelled to a hardware buffer at very high speed (often up to several MB/s). This type of facility can allow data acquisition to proceed at much higher rates than would be possible if each reading had to be individually recorded by the PC. The relatively time-consuming task of transferring data to the PC's memory can then be performed at the end of the data-acquisition sequence. Many DA&C devices allow access to their memory buffers at the same time as new readings are being stored. When sufficient data has been recorded in the hardware buffer, the device's interface circuits generate an interrupt or DMA request in order to initiate transfer to the PC's memory.

The principal benefit offered by hardware buffering is that the DA&C system is not impaired by the variable response times inherent in most PC software. Hardware FIFOs are often essential where a non-deterministic operating system such as Microsoft Windows is used. Because of task switching and associated high interrupt latencies, I/O requests are not always serviced promptly under Windows. Hardware buffers can help to overcome this problem by storing data until the PC is ready to receive it.

Software buffers

The DA&C program itself may also possess its own memory buffers. Such buffers not only supply the decoupling necessary for asynchronous I/O, they can, if carefully implemented, also provide a convenient framework for subsequent data processing. They are usually used for receiving or supplying data during DMA transfers or in interrupt-driven I/O.

Systems employing drivers, or many interacting interrupt handlers, tasks or threads might also make extensive use of temporary buffers.

In an analogue input system, for example, an interrupt handler may place each successive reading in a buffer, from where it can be subsequently retrieved and processed by the main (non-interrupt) portion of the program. This minimizes the processing required within the interrupt handler, allowing it to return quickly and be ready to respond should more data become available. Rapid completion of the interrupt also ensures that lower priority code has the opportunity to run.

Memory buffers can take many forms. We will consider only two basic structures, of which there are a large number of implementations: LIFO buffers and FIFO buffers. All programmers will be familiar with arrays in which each constituent element can be accessed via a numeric index. In high level languages, arrays are used as the basis of various types of buffer. The characteristics of a buffer are determined by the locations in which data is stored and by the order in which it is transferred to and from the buffer.

LIFO buffers

As the name implies, the last item of data to be recorded in a Last-In-First-Out (LIFO) buffer is the first one to be made available when the buffer is read. You should already be familiar with one implementation of LIFO buffers: the 80x86 processor's stack. The usual analogy is that LIFO buffers operate like a pile of books. Just as it is possible to gain access to only the last book placed on the pile (i.e. the one on the top), items of data stored in a LIFO buffer can be retrieved only in the reverse of the order in which they were stored. This property is of limited use in most DA&C systems, but it is occasionally useful if it is necessary to process a sequence of measurements in reverse time order.

Listing 6.2 illustrates two simple C functions that can be used to implement a LIFO buffer. Each element of the buffer is a single 16-bit word, but the example can be readily adapted to handle other data types. The `BufCount` variable should be initialized to zero before storing data in the buffer. If your program reads from or writes to the LIFO buffer from within an interrupt handler, you should disable interrupts whenever non-interrupt code accesses the buffer.

FIFO buffers

Also known as a circular buffer or a ring buffer, the First-In-First-Out (FIFO) buffer is perhaps the most useful buffer structure in DA&C systems. FIFO buffers have many uses in DA&C applications and are essential to facilitate communication between asynchronous processes. They are used as the basis of event-driven systems, for

Listing 6.2 *Accessing a LIFO buffer*

```
unsigned int Buffer[256];
unsigned int BufCount;
     :
     :
void WriteLIFO(unsigned int Data, unsigned char *Full)
{
if (BufCount < 256)
       {
       Buffer[BufCount] = Data;
       BufCount++;
       *Full = 0;
       }
  else *Full = 1;
}

void ReadLIFO(unsigned int *Data, unsigned char *Empty)
{
if (BufCount > 0)
       {
       BufCount--;
       *Data = Buffer[BufCount];
       *Empty = 0;
       }
  else *Empty = 1;
}
```

storing keyboard scan codes and for implementing message queues. They also have many applications in DA&C software: for driver-client interprocess communication, DMA-based I/O and in filtering algorithms. As we shall see in Chapter 8, FIFO buffers are also important features of interrupt-driven serial communications software.

The first item of data recorded in the FIFO buffer is the first one retrieved when the buffer is read. Thus the order in which data is read from the buffer is the same as that in which it was originally stored. FIFO buffers can be visualized as a ring structure such as that shown in Figure 6.3. This example shows only 16 entries in the buffer, but much larger buffers are often used in practice. As the buffer fills, new readings are placed in successive locations around the ring, defined by an index labelled BufIn in the figure. When the buffer is read, the oldest item of data is taken from the tail of the buffer. This is addressed by a second index, BufOut.

Listing 6.3 shows C functions which can be used for reading from and writing to a FIFO buffer. In this example, the buffer is implemented as an array named `Buffer` and has 256 entries. The buffer is managed by means of the two indices `BufIn` and `BufOut`. `BufIn` addresses the next free location in the buffer and `BufOut` points

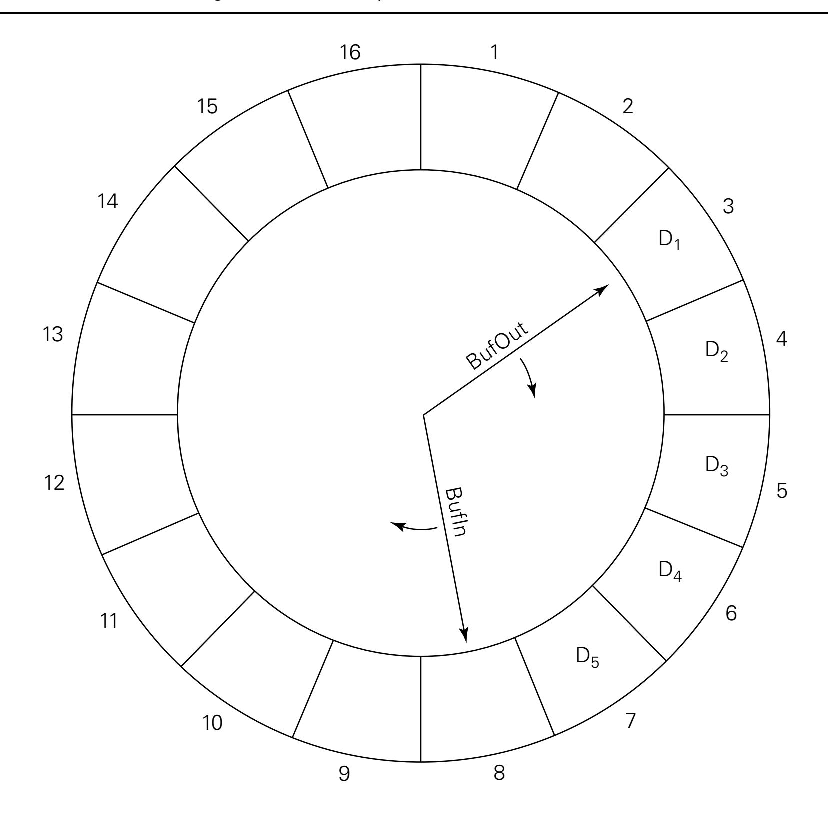


Figure 6.3 *The structure of a FIFO buffer*

to the oldest item of data. Although not shown in the listing, these indices should both be initialized to 0 before accessing the buffer. Likewise, the `BufCount` variable, which is simply a count of the number of readings held within the buffer, should be initialized to 0. Notice that the `BufIn` and `BufOut` indices are incremented until they reach 255 (the end of the `Buffer` array). Subsequent accesses cause the indices to wrap around to the first element in the buffer in order to emulate the structure shown in Figure 6.3.

If the buffer is accessed by two or more asynchronous processes (e.g. from within a hardware interrupt handler and by a non-interrupt routine), calls to the `WriteFIFO()` or `ReadFIFO()` functions will constitute a critical section and must be appropriately protected. You should, for example, disable interrupts when accessing the buffer from non-interrupt code. See Chapter 2 for more on critical sections.

Because memory buffers have a finite (and often quite limited) size it can be easy to run out of space if data is stored at too high a rate, or if the routine that reads the buffer is delayed for some

Listing 6.3 *Accessing a FIFO buffer*

```
unsigned int Buffer[256];
unsigned int BufIn;
unsigned int BufOut;
unsigned int BufCount;
     :
     :
void WriteFIFO(unsigned int Data, unsigned char *Full)
{
if (BufCount < 256)
       {
       Buffer[BufIn] = Data;
       if (BufIn < 255)
              BufIn++;
         else BufIn = 0;
       BufCount++;
       *Full = 0;
       }
  else *Full = 1;
}

void ReadFIFO(unsigned int *Data, unsigned char *Empty)
{
if (BufCount > 0)
       {
       *Data = Buffer[BufOut];
       if (BufOut < 255)
              BufOut++;
         else BufOut = 0;
       BufCount--;
       *Empty = 0;
       }
  else *Empty = 1;
}
```

reason. The programmer has several options when this happens. One possible course of action is to pass an error flag back to the caller, as in Listing 6.3. However, to preserve the relationship between the data stream and the point at which the error occurred, it is often preferable to record the error in the buffer itself. The routine that reads the buffer can then detect the discontinuity in the data stream and take appropriate action.

A third option is simply to record the new data, overwriting the oldest data in the buffer. This may be desirable in certain situations. Some statistical process control applications require the software to maintain a process history of predefined depth (i.e. the *N* most recent sets of readings). This can be easily accommodated by allowing a FIFO buffer to continuously overwrite the oldest data as each new item of data is received.

Another situation where automatic overwriting of data is advantageous is in pre-trigger logging – i.e. where a number of readings must be recorded immediately *prior* to some unpredictable trigger event. An example would be destructive proof testing of steel member. An increasing load may be applied until the member buckles or fractures. The applied load and deformation of the component are measured continuously, but only those readings taken immediately prior to failure may be of interest. The readings can simply be recorded in a FIFO buffer, such that at any given time during the test the buffer holds only the *N* most recent readings. If data acquisition is halted when the component fails, the final contents of the buffer will represent the period leading up to the point of failure.

7 Parallel buses

As far as interfacing to the PC is concerned, it is convenient to divide bus systems into two categories: the PC's internal buses (such as ISA and PCI) and external buses. Although internal buses are an integral part of the PC and a necessary element of all DA&C systems, their operation is largely transparent to the programmer. For this reason, and because they are adequately covered in several books on PC architecture, they will not be described in further detail here. Instead, the present chapter (together with Chapter 8) concentrates on the various external buses that can be used for communicating with devices such as data-logging modules and programmable controllers.

Chapter 8 will deal with serial bus systems, in which data is transferred one bit at a time along a single conductor (or pair of conductors). Parallel buses, which we shall consider in this chapter, possess a separate signal line for each bit. This enables a whole byte, word or double word to be transmitted in one operation, allowing potentially higher data transfer rates.

We will deal with two widely used parallel interfaces: the Centronics parallel port and the IEEE-488 bus (or GPIB). These are of particular interest in PC-based data-acquisition systems. The former is a standard component of virtually all PCs, and there are now a number of parallel-port DA&C devices on the market. The well-known IEEE-488 bus is popular in test and instrumentation applications and is often used for PC-based laboratory interfacing.

This chapter by no means constitutes a comprehensive coverage of parallel bus systems. The popular Small Computer Systems Interface (SCSI) bus, and a number of more specialized backplane buses such as STE and VME, have been excluded. As we have seen in Chapter 1, the latter are used principally for interfacing in industrial DA&C applications. From the PC programmer's perspective they often appear as an extension of the PC's ISA bus. Tooley (1995) provides a useful introduction to these systems. Other bus systems

(such as Metrabyte's MetraBus and the DT-Connect system available from Data Translation Inc.), which are designed specifically for interconnecting components of DA&C systems, have also been excluded because of their proprietary nature.

7.1 Introduction

External parallel buses are usually somewhat simpler in their design than the PC's internal expansion buses. They do not, for example, possess most of the address or control lines that are present on the ISA bus. However, many parallel bus systems do incorporate some form of handshaking in order to strobe data into the receiving device and to control the flow of data across the bus. Handshaking signals used with specific buses are discussed in more detail in the following sections. In contrast to the ISA and PCI buses, some external buses support only 8-bit data transfers.

Most parallel buses operate synchronously – i.e. a common timing or strobe signal is used to synchronize transmission and reception of data. Often, the handshaking signals are automatically generated and sensed by the interface hardware. This relieves the software of the time-consuming burden of having to poll the handshaking lines. An interrupt channel may also be available on the bus, and this allows the interface circuitry to request processor service whenever it is ready to transmit a new byte or whenever new data is received.

Some parallel interfaces operate without the benefit of handshaking or synchronization, and are said to be asynchronous. Because data may arrive at any time, the software must sample the state of the interface frequently enough to accommodate the highest transmission rate. Sampling at too low a rate may result in data bytes being missed. This obviously imposes a considerable overhead on the software. Asynchronous parallel interfaces are employed most often in situations in which the 'data' lines are used, not to carry a byte of data, but instead to sense the state of one or more external devices, such as a limit switch or relay. Interfaces of this nature are more accurately described as a collection of digital control lines rather than a parallel bus. There are now, on the market, a number of parallel digital I/O cards designed for this type of operation. These cards, which are often equipped with isolating circuitry (e.g. relays or opto-isolators), have numerous uses and form an important part of many DA&C systems.

Some parallel interface devices may be suitable for both synchronous and asynchronous communication, depending upon the nature of the software that drives them. For example, the 8255A

Programmable Peripheral Interface, which is used to implement digital I/O on a number of commercial DA&C cards, can be configured for several different operating modes. The Basic I/O mode is suitable for asynchronous digital I/O while more sophisticated modes implement the hardware handshaking features that are necessary to connect to synchronous parallel buses.

7.2 Data acquisition using a parallel bus

The principal benefit of using parallel, rather than serial, buses for data acquisition is that they usually offer significantly higher throughput. As a general rule, most serial buses provide transfer rates of up to about 10 KB/s, whereas a data rate of a few hundred KB/s is achievable with typical external parallel buses (i.e. IEEE-488 and Centronics systems). This speed advantage does not always apply, however. As we will see in Chapter 8, some newer serial bus designs offer the potential for extremely high speed data transfers: up to several tens of MB/s!

One of the most serious restrictions imposed by parallel buses is that they are mostly designed for use with relatively short cables. Unless fibre optic links are employed, this precludes their use for communicating with remote data loggers and similar systems. It is not usually advisable to employ cables longer than about 1 metre with buses driven directly from TTL devices such as an 8255A Programmable Peripheral Interface (PPI). Up to about 2 to 3 metres of good quality shielded cable may normally be used in conjunction with the Centronics parallel port, while the IEEE-488 bus supports a total cable length of not more than 20 m. This compares with distances of up to several thousand metres that are permissible with some serial interfaces. The maximum practicable transmission distance with any parallel bus does of course depend upon the impedance of the cable and the rate at which data is to be transmitted. The degree of coupling between the bus lines may also be an important consideration. Slow transmission rates can, in some cases, permit slightly longer cables to be used.

Most parallel systems employ a ‘multi-drop’ bus topology – i.e. several devices connected in parallel to the same data and control lines. A good example of this is the IEEE-488 (GPIB) bus which we will discuss later in this chapter. Point-to-point topologies are also sometimes used. This configuration is often employed with devices connected to the PC’s parallel (Centronics) port.

Parallel buses are used in a great variety of data-acquisition systems. Their principal role is for high speed communication with laboratory

test equipment and instruments such as digital voltmeters, frequency counters or logic analysers. A number of products are available which make use of, for example, the PC's Centronics port to interface directly to an ADC. When used in conjunction with suitable line drivers, relays, or opto-isolators, parallel interfaces can also be used in industrial systems to interface to Programmable Logic Controllers (PLCs), control panels, indicators, motor drives and a multitude of other devices.

7.3 The PC's parallel port

Almost all PCs are equipped with at least one parallel port, but most machines will accommodate up to three separate ports. The parallel port was designed specifically for interfacing to printers. The terminology used to describe the various connector pins and signals reflects this. On some systems the parallel port may be used for other purposes, such as connecting to external disk drives, tape devices or to copy-protection keys (dongles). It also provides a convenient means of interfacing to data-acquisition and/or control systems. We will not discuss in detail how to drive a printer via the parallel port – it is normally preferable to use the operating system or BIOS services that are provided for this purpose (see, for example, the texts by Norton and Wilton (1988), Phoenix Technologies Ltd (1989) or Dettmann and Johnson (1992)). Instead, this section will concentrate on the operation of the parallel port's hardware and will discuss how it can be programmed for use in DA&C applications.

Parallel port standards

Modern PCs are equipped with parallel ports conforming to a variety of standards. There are four basic classes of parallel port:

1. The standard unidirectional port: present on IBM PC, XT and AT machines.
2. The bidirectional port which was introduced in the IBM PS/2 range.
3. The Enhanced Parallel Port (EPP) developed by Xircom Inc., Intel and Zenith Data Systems.
4. The Enhanced Capabilities Port (ECP) developed by Hewlett Packard and Microsoft.

The standard parallel port was designed primarily for unidirectional output. As such, it possesses only one 8-bit output port and a group of five digital input lines. The latter usually carry control information,

but in some applications they provide a means of inputting data from external devices. Data is usually read one nibble (4 bits) at a time: the fifth input line carries control or interrupt signals.

The bidirectional parallel port is present on the IBM PS/2 range and on some older AT 'clone' machines. For compatibility with earlier systems, this port emulates the standard unidirectional port by default. However, it can be switched, by software, to an input mode, allowing its 8-bit data port to receive a byte of information from a peripheral device.

More modern ISA/PCI machines are equipped with an Enhanced Parallel Port (EPP) which is a further extension of the standard parallel port. This type of system employs a bidirectional data bus, but also carries out the data transfer handshake automatically as soon as the software writes data to the port. This removes the burden of handshaking from the software and allows a byte to be transferred in only one I/O cycle. At least four `OUT` or `outportb()/outp()` instructions would be required for a software-controlled handshaking sequence using a standard parallel port. The EPP can, of course, emulate a standard parallel port if the high speed data transfer capability is not required. To maintain compatibility with the standard port, the EPP defaults to this emulation mode when power is first applied. The enhanced high speed mode may subsequently be activated by software. A number of the parallel port's connector pins ($\overline{\text{STROBE}}$, $\overline{\text{AUTOFEED}}$, and $\overline{\text{SELECT-IN}}$: see *Connector pin assignment* later in this chapter) are used for different purposes when the EPP's enhanced mode is activated, although they revert to their normal function in the default standard mode. The EPP is used on some portable computers to circumvent their limited expansion capability and to provide a means of interfacing them to peripherals other than printers.

The ECP provides similar facilities to those of the EPP, but, in addition, implements data compression and error detection facilities as well as an addressing scheme that allows a single port to address one of up to 128 separate I/O devices.

The IEEE-1284 (1994) standard encompasses all four classes of parallel port and defines every aspect of the parallel port interface. It reclassifies the previous port designs as separate modes of a new type of port. This standard is becoming widely adopted for interfacing to peripherals and to some DA&C devices, but there are still a very large number of the older port designs in use.

Most data-acquisition applications do not require the very high rates of throughput possible with the EPP, ECP and IEEE-1284 ports. In the remainder of this chapter, we will concentrate on the basic features offered by the unidirectional and bidirectional parallel

ports or modes. Unless specified to the contrary, the following text excludes any discussion of the more advanced features of EPP, ECP and IEEE-1284. Remember, however, that these standards maintain backward compatibility with the earlier devices and so the information provided will also be of use on modern IEEE-1284 compliant machines. Further information on the EPP may be found in the texts by van Gilluwe (1994) and Buchanan (1999). Rosch (1996) also provides a detailed account of the various parallel port standards.

Data acquisition via the parallel port

The parallel port offers several advantages for DA&C. First, it is cheap to use – it is a standard component of all PCs – and it is often only necessary to purchase or construct a suitable connector and cable. Also, the computer can be easily unplugged from the external device: there is no need to insert special adaptor cards in the PC's expansion slots. This is a particularly relevant consideration when the number of expansion slots is limited (e.g. when using a portable PC). Finally, and often most importantly, the parallel port offers the potential for quite high speed data transfer.

Speeds of up to about 150 KB/s are possible on a standard unidirectional parallel port, although the actual maximum data transfer rate will, of course, depend upon the speed of the controlling software and upon the response time of the device attached to the port. Most printer interfaces, for example, are driven at a fraction of the maximum rate: perhaps 10 KB/s or less. Some new versions of the parallel port, conforming to the EPP standard or the more recent IEEE-1284 standard, are capable of transmitting data at up to 2 MB/s, although it is difficult in practice to sustain data rates of more than about 800 KB/s.

A number of manufacturers now produce DA&C modules which connect directly to the PC's parallel port. Some devices are very simple and inexpensive, incorporating, for example, a single channel 8-bit ADC. Others provide a more comprehensive set of features: multiplexed analogue input, multi-channel analogue output, digital I/O or complex counter/timer devices for digital pulse and frequency measurement.

The main disadvantage with using a parallel port for data acquisition is that cable lengths must be limited to less than about 1.5 to 3 m, depending upon port design and cable quality. Transmission distance can be extended by using fibre optic adaptors.

A further limitation is that the port provides only a small number of I/O lines. There are five input lines on the standard unidirectional

parallel ports and this may be inadequate in some applications. The parallel ports present on a few older clone machines do not even conform to the basic unidirectional port standard and have an even smaller number of active input lines! Some peripheral devices (most notably copy protection 'dongles') circumvent this limitation by transferring data bits in a serial manner, using just one of the available I/O lines. This does negate the parallel port's speed advantage and complicates programming somewhat. In the absence of bidirectional, EPP or ECP ports, the most satisfactory means of increasing the number of I/O lines and of implementing bidirectional data transfers is to interface the port to a device such as an 8255A PPI via non-inverting octal buffers and suitable logic.

Parallel port addresses

Each parallel port appears to the programmer as a set of three registers in the PC's I/O space. The starting (or base) address of each register group is recorded by the BIOS's POST routines in a four-word table at address 0040:0008h in the BIOS Data Area. This is shown in Table 7.1. The total number of parallel ports present in the system is stored as a binary-coded number in bits 14 and 15 of the word at 0040:0010h in the BIOS Data Area.

The IBM PC and XT, and compatible machines, will accommodate up to four separate parallel ports. All four of the above locations may be occupied on these systems. However, on the IBM AT and modern PCs, the location previously used to hold the fourth parallel port address (i.e. 0040:000Eh) is reserved. On the PS/2 range of machines (and some AT compatibles) this location contains the segment address of the Extended BIOS Data Area. The parallel port base addresses that are normally used on the various models of PC and PS/2 are listed in Table 7.2. As there can be some variation

Table 7.1 *Parallel port address table in the BIOS Data Area*

Address	*Contents*
0040:0008h	I/O address of first parallel port.
0040:000Ah	I/O address of second parallel port (or 0 if less than 2 ports present).
0040:000Ch	I/O address of third parallel port (or 0 if less than 3 ports present).
0040:000Eh	IBM PC, XT: I/O address of fourth parallel port (or 0 if not present). IBM AT: Reserved. IBM PS/2: Segment address of extended BIOS Data Area.
0040:0010h	Bits 14 and 15 hold the number of parallel ports detected by the BIOS.

Table 7.2 *Usual parallel port addresses*

Parallel port	*Base address on PC, XT*	*Base address on AT*	*Base address on MCA systems*
1	3BCh or 378h	378h	3BCh
2	378h or 278h	278h	378h
3	Undefined	Undefined	278h

between the various 'compatible' machines, it is prudent to obtain the port's base address from the BIOS Data Area rather than to code the address into your program.

Note that the BIOS printer services obtain the parallel port addresses from the BIOS Data Area and, if all parallel-port driver software is designed to do likewise, it is then very simple to redirect I/O operations to a different port by simply rearranging the contents of the address table.

The structure of the parallel port

Although the parallel port is a fairly simple device, there are a number of difficulties associated with using it for two-way data interchange. Before considering the topic of communication we will first discuss the parallel port's structure and method of operation.

Overview

Figure 7.1 is a schematic representation of the structure of the parallel port. Each parallel port contains three registers which occupy contiguous addresses in the PC's I/O space. Actually, read and write operations performed on two of these I/O addresses (i.e. the Data and Control Register addresses) cause different internal registers to be accessed. However, most of the bits within each pair of registers are mapped to the same signal lines and, for this reason, it is more convenient to think of reading and writing operations as accessing the same register.

The majority of the bits that can be addressed via these registers are used to directly control or sense the state of the various signal pins present on the connector (see the following section for a list of pin connections). In most cases, a logical 1 bit corresponds to a high voltage (+5 V) at the associated connector pin, but the $\overline{\text{SELECT-IN}}$, $\overline{\text{AUTOFEED}}$, $\overline{\text{STROBE}}$ and BUSY lines are inverted as shown in Figure 7.1. Other bits present in the various registers are used to enable or disable interrupts and, on bidirectional ports, for selecting

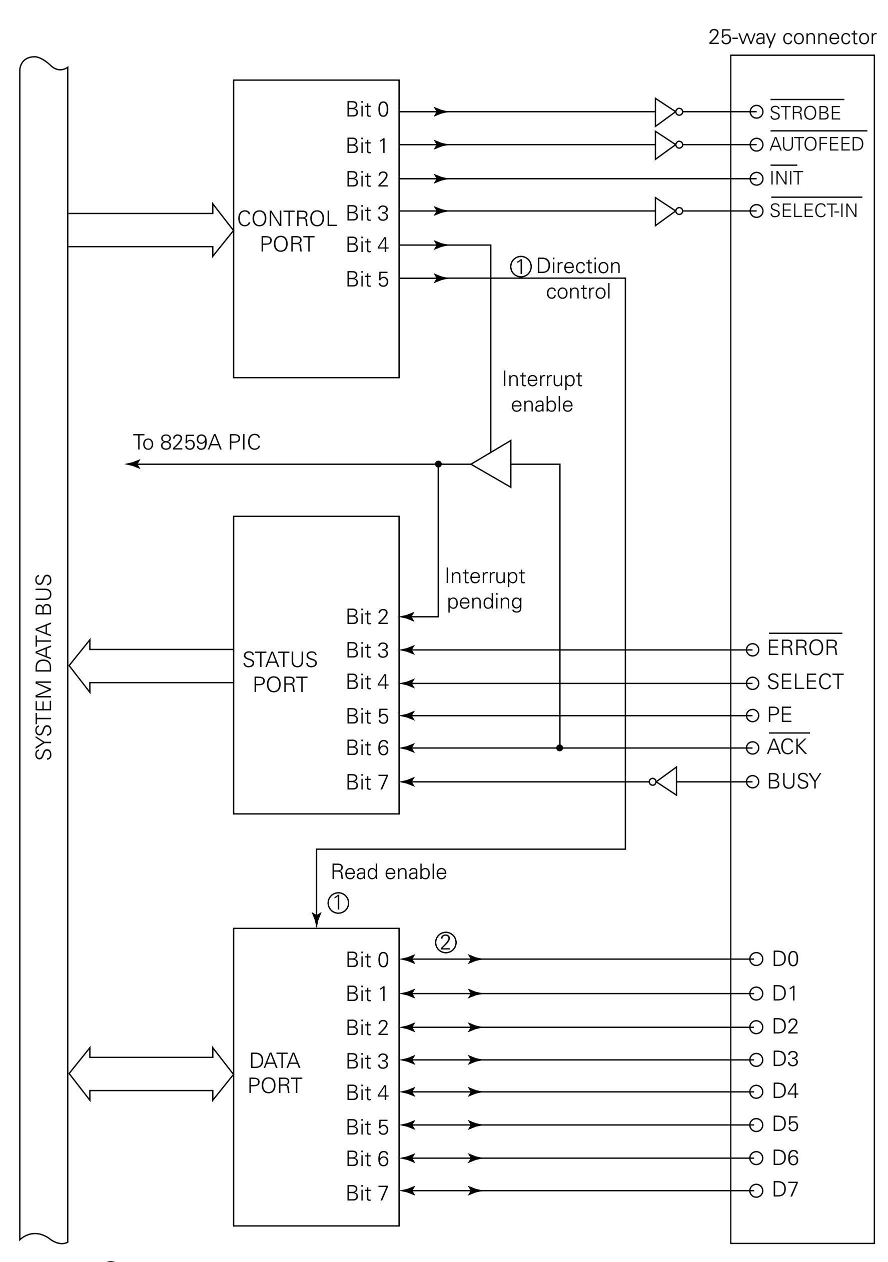


Figure 7.1 *Schematic representation of the parallel port*

the direction of data transfer. Note that the overstrike (e.g. in $\overline{\text{STROBE}}$) indicates only that the signal is active, or asserted, when at a logic-low level: it is not meant to indicate that the signal is inverted between the Status or Control Register and the connector pin.

The standard unidirectional parallel port, does not allow data to be input via the Data Register. However, the bidirectional type of parallel port can be programmed (via bit 5 of the Control Register) to permit both input and output via the Data Register. Listing 7.1, shown later in this chapter, includes a function which illustrates how to determine whether the parallel port hardware supports this 'extended' mode.

The $\overline{\text{ACK}}$ input line may be sensed via bit 6 of the Status register. As shown in Figure 7.1, this line can also be used to generate interrupts. The falling edge of a pulse on $\overline{\text{ACK}}$ will cause an interrupt to occur, but only if bit 4 of the Control Register is set. The 8259 PIC's interrupt mask must also have been modified in order to enable interrupts on the appropriate IRQ line. The first parallel port is usually assigned to IRQ7 and the second to IRQ5. No specific interrupt levels are reserved for other parallel ports which might be present in the system. In these cases it is usual to configure the port to use any free interrupt channel. The IRQ level may usually be selected by means of a jumper or DIP switch. Once an interrupt signal has occurred on the $\overline{\text{ACK}}$ line, bit 2 of the Status Register indicates that an interrupt is pending. Note that the BIOS's printer services do not make use of the parallel port's interrupt facilities, although some Windows EPP or ECP drivers do.

Connector pin assignment

The PC's parallel port employs a female 25-way D-type connector. This usually connects to a printer via a cable terminated with a male 36-way Amphenol connector. The pin assignments for both connector types are listed in Table 7.3.

Registers and programming details

The Standard parallel port has three registers: the Data Register, the Status Register and the Control Register. These are also supported by the more advanced implementations of the parallel port (e.g. IEEE-1284 compliant ports).

The Data Register (offset 0, R/W)

This is normally used for sending 8-bit characters to a printer, but in DA&C applications it may also be used for sending out commands, data or other signals to data-logging or control units.

Table 7.3 *Parallel port connector pin assignments*

Pin number		
25-way D-type	*36-way Amphenol*	*Signal*
1	1	$\overline{\text{STROBE}}$
2	2	D0
3	3	D1
4	4	D2
5	5	D3
6	6	D4
7	7	D5
8	8	D6
9	9	D7
10	10	$\overline{\text{ACK}}$
11	11	BUSY
12	12	PE
13	13	SELECT
14	14	$\overline{\text{AUTOFEED}}$
15	32	$\overline{\text{ERROR}}$
16	31	$\overline{\text{INIT}}$
17	36	$\overline{\text{SELECT-IN}}$
18–25	19–30, 33	Signal ground
–	15	Not connected
–	16	0 V (logic ground)
–	17	Chassis ground
–	18	Not connected
–	34	Not connected
–	35	Logic 1

On the standard parallel port, or on the bidirectional port when read mode is disabled (Control Register, bit 5 = 0), all bytes written to the Data Register are latched so that the data remains on the corresponding connector pins. Any subsequent read operations will return the last byte written to the register. Note that reading the Data Register will return the data previously latched: it is not possible to read the state of the connector's D0–D7 pins on the standard unidirectional parallel port.

When data reads have been enabled (Control Register, bit 5 = 1), the data output latch is isolated from the connector pins so that any bytes written to the Data Register are prevented from reaching the parallel port connector. In this mode, it is possible to sense the state of the D0–D7 connector pins by *reading* the Data Register.

Table 7.4 *The Status Register*

Bit	*Description*
0	Unused/reserved.
1	Unused/reserved.
2	Interrupt request (IRQ) pending on MCA systems. Unused on non-MCA systems.
3	$\overline{\text{ERROR}}$ line status (1 = +5 V nominal).
4	SELECT line status (1 = +5 V nominal).
5	PE line status (1 = +5 V nominal).
6	$\overline{\text{ACK}}$ line status (1 = +5 V nominal).
7	BUSY line status – inverted (0 = +5 V nominal).

The Status Register (offset 1, R/O)

This register is normally used for reading the status of an attached printer. Bits 3 to 7 of the Status register reflect the state of the five input lines listed in Table 7.4. Note that the BUSY line is inverted so that a high voltage (+5 V) on the connector pin will result in a zero BUSY bit. As mentioned previously, a low pulse on the $\overline{\text{ACK}}$ line can be made to generate an interrupt if required. On a bidirectional parallel port, bit 2 indicates whether an interrupt is pending.

The Control Register (offset 2, R/W)

When a printer is connected to the parallel port, the Control Register is normally used to control data transfers to the printer. This is accomplished by means of four digital output lines which can be manipulated via the four low order bits of the Control Register. When interfacing to equipment other than a printer, these lines can be used for a variety of different purposes. The $\overline{\text{STROBE}}$, $\overline{\text{AUTOFEED}}$ and $\overline{\text{SELECT-IN}}$ lines are inverted so that each bit must be set to 0 in order to generate a high (+5 V) voltage at the corresponding connector pin. However, the $\overline{\text{INIT}}$ output line is not inverted. The four output lines are all latched so that, once written, the same bit pattern will normally remain on the connector pins. Reading from this register will return the values previously written to these lines.

Two other bits are also present in the Control Register. These are used for enabling the parallel port interrupt and, on a bidirectional parallel port, for controlling the direction of data flow through the Data Register. Table 7.5 lists the bits present in this register.

Table 7.5 *The Control Register*

Bit	*Write*	*Read*
0	$\overline{\text{STROBE}}$ pin (0 = +5 V nominal).	$\overline{\text{STROBE}}$ pin status (0 = +5 V nominal).
1	$\overline{\text{AUTOFEED}}$ pin (0 = +5 V nominal).	$\overline{\text{AUTOFEED}}$ pin status (0 = +5 V nominal).
2	$\overline{\text{INIT}}$ pin (1 = +5 V nominal).	$\overline{\text{INIT}}$ pin status (1 = +5 V nominal).
3	$\overline{\text{SELECT-IN}}$ pin (0 = +5 V nominal).	$\overline{\text{SELECT-IN}}$ pin status (0 = +5 V nominal).
4	0 = Disable parallel port interrupt. 1 = Enable parallel port interrupt.	Current interrupt-enable status.
5	0 = Write via Data register enabled (standard/compatibility mode). 1 = Read via Data register enabled (write via Data register disabled).	Unused/reserved.
6	Unused/reserved.	Unused/reserved.
7	Unused/reserved.	Unused/reserved.

Driving a printer via the parallel port

So far we have seen how each control and status line present in the parallel port is mapped to the various registers, but we have refrained from discussing the mechanisms used to transfer data to a printer. This information is, of course, superfluous if the parallel port is to be used for interfacing to devices such as relays, stepping motors or data-logging equipment. However, if it is necessary to interface to a printer, or to a DA&C device which operates in a similar way, it is important to understand the basic principles of the data transfer sequence involved.

Table 7.6 indicates how the various control and status signals are used to control a printer. Normally, the printer-driving software will force the $\overline{\text{SELECT-IN}}$ line low to select the printer. This may occur once only, perhaps at the beginning of a program. The printer will subsequently set the SELECT line high. To transfer each character, the following sequence of events occurs:

1. The software waits until the printer's BUSY signal goes low, which indicates that the printer is ready to receive a character.
2. The software places a character code on the D0–D7 lines and, after a short delay pulses the $\overline{\text{STROBE}}$ line low. The falling edge

Table 7.6 *Printer control and status signals*

Pin/signal	*Direction*	*Description*
BUSY	Input	High when the printer is busy and unable to accept any further data. Goes low when ready to receive more data.
$\overline{\mathrm{ACK}}$	Input	Pulses low to acknowledge receipt of data.
$\overline{\mathrm{STROBE}}$	Output	Pulses low to indicate that valid data is present on D0–D7. The printer must read D0–D7 when it detects the $\overline{\mathrm{STROBE}}$ pulse.
PE	Input	High when the printer has run out of paper.
SELECT	Input	High when the printer is selected and active.
$\overline{\mathrm{ERROR}}$	Input	Low when the printer detects a paper out (PE) error condition, when the printer is off line, or when some other error is detected.
$\overline{\mathrm{SELECT\text{-}IN}}$	Output	Low selects the printer. This signal is ignored on some printers.
$\overline{\mathrm{INIT}}$	Output	Low pulse, lasting at least 50 μs, initializes the printer.
$\overline{\mathrm{AUTOFEED}}$	Output	Low causes the printer to automatically generate a Line Feed character immediately after receiving each Carriage Return character. This signal is ignored by some printers. The auto-line feed facility is often selectable via the printer's DIP switches or front panel.

of the $\overline{\mathrm{STROBE}}$ pulse causes the printer to immediately set the BUSY line high and then to read the data from the D0–D7 lines.

3. When the printer has read and stored the data, it pulses the $\overline{\mathrm{ACK}}$ line low in order to acknowledge receipt of the data. As $\overline{\mathrm{ACK}}$ returns to a high state, the printer pulls the BUSY line low again to signal that it is ready to receive the next character.

The $\overline{\mathrm{ACK}}$ pulse can be made to generate an interrupt. Using this facility, you can install an interrupt handler to transfer a series of characters from a memory buffer to the printer.

The PC may pulse the $\overline{\mathrm{INIT}}$ line at any time to reset the printer. The driving software should monitor the PE and $\overline{\mathrm{ERROR}}$ lines in order to detect error conditions such as the printer running out of paper or being switched off line. Many different types and models of printer can be connected to the parallel port. Most have stable and noise-free interfaces, but in some cases electrical noise, caused by badly shielded or grounded cables, may be problematic. When writing interface software to sense the condition of the $\overline{\mathrm{ACK}}$, BUSY, PE, SELECT and $\overline{\mathrm{ERROR}}$ lines it is advisable to sample the relevant bits in the Status Register at least two or three times. This reduces

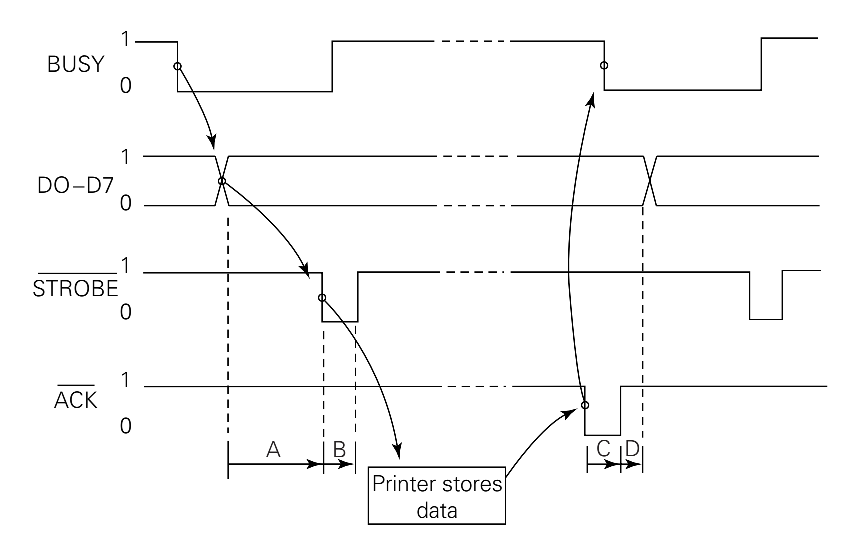


Figure 7.2 *Handshake sequence for data transfer via the parallel port to a printer*

the likelihood that spurious noise spikes will disturb the handshake sequence. The data transfer handshake is illustrated in Figure 7.2.

The timing specification for the transfer is only loosely defined, particularly in the case of older hardware designs. The minimum delay times required to transfer data to a fast printer are A = B = C = D = 0.5 μs. Some printers may require the various signals to be held for a greater length of time. Sanchez and Canton (1994) recommend that the $\overline{\text{STROBE}}$ pulse should last for 5 μs or more. Buchanan (1999) gives similar figures while the IEEE monographs by Maine (1986) and Marnham (1994) specify the following minimum timings:

A. $\overline{\text{STROBE}}$ pulse delay 50 μs
B. $\overline{\text{STROBE}}$ pulse period 1 μs
C. $\overline{\text{ACK}}$ pulse period 100 ns
D. Delay after $\overline{\text{ACK}}$ before removing data 10 μs

The variation in the quoted timing figures reflects the loosely defined standards adopted by early parallel port implementations. According to Rosch (1996), the more rigorous IEEE-1284 standard's Compatibility mode (which emulates a unidirectional port) specifies a $\overline{\text{STROBE}}$ pulse period (B) of 0.5–500 μs and an $\overline{\text{ACK}}$ pulse period (C) of 0.5–10 μs.

A simple parallel port driver

Listing 7.1 is an example of a basic parallel port driver which provides access to the various I/O lines present at the connector. The listing consists simply of a library of (almost) independent C routines that can be called to perform specific tasks. Functions are included to determine the address of each parallel port in the system and to check whether the ports are of the bidirectional type.

To use this driver, the caller must first invoke the `SearchForLPTPorts()` function. This will initialize the array of `LPT` structures according to the number, type and location of LPT (i.e. parallel) ports found. The caller may then examine the `BaseAddr` and `ExtMode` fields of each element in the array to determine whether the corresponding parallel port is available and, if so, whether it supports the so-called 'extended' (read) mode of the bidirectional port. Thereafter, the remaining functions contained within the listing can be called as and when needed to read or write data via the parallel port. Each function is individually documented and its purpose should be self-explanatory.

The driver automatically inverts the states of the $\overline{\text{SELECT-IN}}$, $\overline{\text{AUTOFEED}}$, $\overline{\text{STROBE}}$ and BUSY signals so that a high bit passed between the calling routine and the driver functions always corresponds to a high voltage (+5 V) at the corresponding connector pin. When using this driver, the programmer need not be concerned with the locations of each bit within the various registers: all I/O

Listing 7.1 *A parallel port software driver*

```
/*                  Bidirectional Parallel Port Driver
                    ----------------------------------

 This driver allows access to the three parallel port registers. The connector
 pins corresponding to the various bits in the bit patterns passed to/from
 these driver procedures are mapped as follows:

 ----------------------------------------------------------------------
   Port               Bit pattern passed to or from driver procedures
                        7     6     5     4     3     2     1     0
 ----------------------------------------------------------------------
   Data port           D7    D6    D5    D4    D3    D2    D1    D0
   Status port         ---   ---   ---  BUSY   ACK   PE   SLCT  ERROR
   Control port        ---   ---   ---   ---  SL-IN INIT   AFD  STROBE
 ----------------------------------------------------------------------

 All high bits passed as arguments to the driver procedures correspond to
 logical high signals at the corresponding connector pins - i.e. the software
 compensates for the logical inversion of some of the LPT port lines (BUSY,
 -SL-IN, -AFD and -STROBE are all inverted in hardware and this is compensated
 for by the software).
```

Listing 7.1 *(continued)*

```
 The driver allows individual bits in the data port or control port to be set
 without disturbing any other bits in the port. It also allows the bit pattern
 of the whole port to be changed in one operation. The five status lines may
 also be read in one operation.


 Extended read mode can be enabled (if supported) to allow read operations to
 be performed via the data port.


 The -ACK line can be used to generate an interrupt whenever it pulses low.
 The interrupt can be enabled or disabled as required using this driver
 (although code for manipulating the 8259 PIC and for intercepting the
 interrupt is not included).
*/

#include <dos.h>

#define MaxNumLPTPorts 3
#define True           1
#define False          0

/* =========================== Data Declarations =========================== */

struct LPTPortRec
  {
  unsigned int  BaseAddr;          /* Base address of parallel port hardware */
  unsigned char ExtMode;                    /* >0 if extended mode supported */
  unsigned char LastData;          /* Last data output via the Data register */
  unsigned char LastCtrl;       /* Last data output via the Control register */
  };


struct LPTPortRec LPT[MaxNumLPTPorts];        /* One structure for each port */



/* ========================== Function Prototypes ========================== */

unsigned int LPTPortBaseAddress(unsigned char Port);
unsigned char ExtendedModeSupported(unsigned char Port);
void SearchForLPTPorts(void);
void WriteData(unsigned char Port, unsigned char Data);
unsigned char ReadData(unsigned char Port);
unsigned char ReadStatus(unsigned char Port);
void WriteCtrl(unsigned char Port, unsigned char Data);
void SetDataBit(unsigned char Port, unsigned char BitNum, unsigned char High);
void SetCtrlBit(unsigned char Port, unsigned char BitNum, unsigned char High);
void SetExtendedMode(unsigned char Port, unsigned char Enable);
void SetACKInterrupt(unsigned char Port, unsigned char Enable);
void InitializeLPTPort(unsigned char Port);



/* ======================= Function Implementations ======================== */


unsigned int LPTPortBaseAddress(unsigned char Port)
```

Listing 7.1 *(continued)*

```
/* Returns the base address of the specified LPT port. The Port parameter is
   zero based. */
{
return peek(0x40,(0x08 + (2 * Port)));
}


unsigned char ExtendedModeSupported(unsigned char Port)
/* Determines whether the specified LPT port supports extended read mode. */
{
unsigned char CtrlPort;
unsigned char BitPtn;
unsigned char Supported;

CtrlPort = inportb(LPT[Port].BaseAddr+2);         /* Get control port status */
outportb(LPT[Port].BaseAddr+2,(CtrlPort | 0x20));     /* Try to activate the */
                                                       /* Extended mode */
/* Check whether we can still read back data */
Supported = False;
BitPtn = 0x00;
do
  {
  outportb(LPT[Port].BaseAddr,BitPtn);
  if (inportb(LPT[Port].BaseAddr) != BitPtn) Supported = True;
  BitPtn++;
  }
while (BitPtn != 0xFF);

outportb(LPT[Port].BaseAddr+2,CtrlPort);    /* Restore original control port */
return Supported;
}


void SearchForLPTPorts()
/* Searches through the BIOS data area locations at offsets 08h, 0Ah and 0Ch
   to determine the addresses of LPT1, LPT2 and LPT3 ports. A value of zero in
   any one of these locations indicates that no corresponding parallel port is
   available. This function checks whether each port supports extended mode
   (i.e. bidirectional data transfer). */
{
unsigned char Port;

for (Port = 0; Port < MaxNumLPTPorts; Port++)
  {
  LPT[Port].BaseAddr = LPTPortBaseAddress(Port);
  if (LPT[Port].BaseAddr != 0)
         LPT[Port].ExtMode = ExtendedModeSupported(Port);
    else LPT[Port].ExtMode = False;
  }
}


void WriteData(unsigned char Port, unsigned char Data)
/* This function writes the specified Data byte to the data register of the
   LPT port specified by Port. A low bit corresponds to a logical low signal
   on the corresponding connector pin. */
```

Listing 7.1 *(continued)*

```
{
LPT[Port].LastData = Data;
outportb(LPT[Port].BaseAddr,Data);
}


unsigned char ReadData(unsigned char Port)
/* This reads the data port if extended mode is supported and data reads are
   enabled (via the Direction Control bit in the control port). If reads are
   not possible, this function returns the last data written to the control
   port. A low bit in Data corresponds to a logical low signal at the
   corresponding connector pin. */
{
if ((LPT[Port].ExtMode) && ((LPT[Port].LastCtrl & 0x20) == 0x20))
       return inportb(LPT[Port].BaseAddr);
  else return LPT[Port].LastData;
}


unsigned char ReadStatus(unsigned char Port)
/* Reads the Status port lines and returns them, coded as follows (MSB first):
   BUSY, -ACK, PE, SLCT and -ERROR. A low bit corresponds to a logical low
   signal on the corresponding connector pin. */
{
return (((inportb(LPT[Port].BaseAddr+1) ^ 0x80) >> 3) & 0x1F);
}


void WriteCtrl(unsigned char Port, unsigned char Data)
/* This function writes the low order four bits of Data to the Control register
   leaving the Interrupt Enable and Direction Control bits unchanged. The four
   bits are, in order from MSB to LSB: -SL-IN, -INIT, -AFD and -STROBE. A low
   bit corresponds to a logical low signal on the corresponding connector
   pin. */
{
LPT[Port].LastCtrl = ((Data ^ 0x0B) & 0x0F) | (LPT[Port].LastCtrl & 0xF0);
outportb(LPT[Port].BaseAddr+2,LPT[Port].LastCtrl);
}

void SetDataBit(unsigned char Port, unsigned char BitNum, unsigned char High)
/* Sets the state of a single bit (BitNum = 0 to 7) in the specified LPT port's
   data port. If High is True, the corresponding connector pin is set to a
   logical high state. */
{
unsigned char Mask;

Mask = 0x01 << (BitNum % 8);
if (High)
       LPT[Port].LastData = LPT[Port].LastData | Mask;
  else LPT[Port].LastData = LPT[Port].LastData & ~Mask;
outportb(LPT[Port].BaseAddr,LPT[Port].LastData);
}


void SetCtrlBit(unsigned char Port, unsigned char BitNum, unsigned char High)
/* Sets the state of a single bit (BitNum = 0 to 3) in the specified LPT port's
   control port. If High is true, the corresponding connector pin is set to a
   logical high state. */
```

Listing 7.1 *(continued)*

```
{
unsigned char Mask;

Mask = 0x01 << (BitNum % 4);
LPT[Port].LastCtrl = LPT[Port].LastCtrl ^ 0x0B; /* Uninvert bits in LastCtrl */
if (High)
       LPT[Port].LastCtrl = LPT[Port].LastCtrl | Mask;
  else LPT[Port].LastCtrl = LPT[Port].LastCtrl & ~Mask;
LPT[Port].LastCtrl = LPT[Port].LastCtrl ^ 0x0B; /* Reinvert bits in LastCtrl */
outportb(LPT[Port].BaseAddr+2,LPT[Port].LastCtrl);
}

void SetExtendedMode(unsigned char Port, unsigned char Enable)
/* Enables or disables the parallel port's extended mode (if available). This
   procedure has no effect if the port does not support extended mode. */
{
if (LPT[Port].ExtMode)
  {
  if (Enable)
         LPT[Port].LastCtrl = LPT[Port].LastCtrl | 0x20;
    else LPT[Port].LastCtrl = LPT[Port].LastCtrl & 0xDF;
  outportb(LPT[Port].BaseAddr+2,LPT[Port].LastCtrl);
  }
}


void SetACKInterrupt(unsigned char Port, unsigned char Enable)
/* Enables or disables the parallel port's interrupt. */
{
if (Enable)
       LPT[Port].LastCtrl = LPT[Port].LastCtrl | 0x10;
  else LPT[Port].LastCtrl = LPT[Port].LastCtrl & 0xEF;
outportb(LPT[Port].BaseAddr+2,LPT[Port].LastCtrl);
}

void InitializeLPTPort(unsigned char Port)
/* Sets all outputs to logical low levels and disables the parallel port
   interrupt and extended mode (if available). */
{
WriteData(Port,0x00);
WriteCtrl(Port,0x00);
SetExtendedMode(Port,False);
SetACKInterrupt(Port,False);
}
```

lines are mapped to the low order bits of each register as noted in the listing.

7.4 The IEEE-488 (GPIB) bus

The IEEE-488 bus standard is also known as the General Purpose Interface (or Instrument) Bus or GPIB. It originates from the HP-IB bus originally developed by Hewlett Packard in the mid-1960s. It

was adopted by the Institute of Electrical and Electronics Engineers (IEEE) as the basis of a new standard for parallel communications designated IEEE-488. This was revised in 1978 and updated again in 1987. These two revised standards are often referred to as IEEE-488.1 and IEEE-488.2 respectively, the latter maintaining backward compatibility with the earlier standard. The original IEEE-488 specification relates mainly to the hardware elements of the bus. IEEE-488.2, however, is concerned more with command protocols, defining such things as the order of multiple bus commands and transaction timeouts. Error handling and status reporting were also standardized along with some commonly used commands and data structures. In the remainder of this chapter we will refer to both standards simply as 'IEEE-488' except where discussing specific differences between them.

The IEEE-488 bus was originally used for interfacing to laboratory test equipment (e.g. frequency meters, spectrum analysers, calorimeters, logic analysers etc.) and to printers or plotters. Today the bus has become very popular in both manufacturing and research environments, and a great diversity of instruments are equipped with IEEE-488 interfaces. It is now possible to connect many common and relatively inexpensive measuring instruments – digital voltmeters, for example – to the IEEE-488 bus.

Overview of the IEEE-488 bus

The IEEE-488 standard allows up to 15 devices (including the PC) to be connected together on the same party-line bus as illustrated in Figure 7.3. The total length of the interconnecting cables must not exceed 20 m and the distance between any two bus devices must be no more than 2 m.

Each of the 15 possible devices is assigned a unique address in the range 0 to 30. This is known as the primary address and

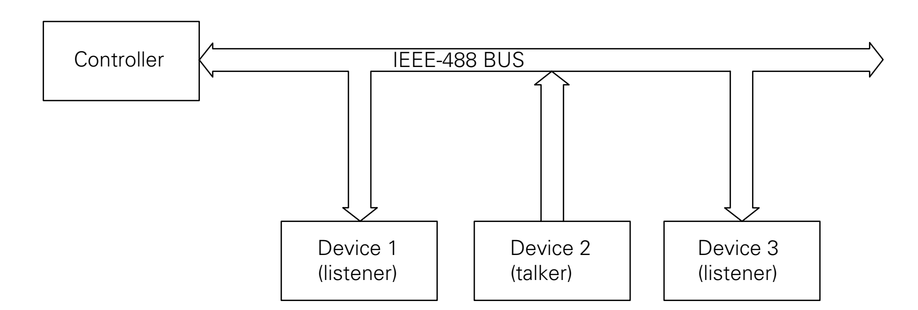


Figure 7.3 *IEEE-488 bus topology*

is usually configured by means of a DIP switch or an analogous programmable facility. Each bus device may also incorporate up to 32 sub-units which are capable of operating independently of each other. These sub-units may be individually addressed using secondary addresses in the range 0 to 31. The sub-units within each bus device consist of logically independent (although not necessarily physically separate) units. Secondary address allocation is generally device specific. In some cases, the secondary addresses are used to select specific features or data processing modes of a single unit. One secondary address may, for example, be reserved for receipt of configuration commands, while another is reserved for receiving operational commands. Alternatively, a device connected to multiple sensors might use different secondary addresses to configure and access each sensor.

As indicated in Figure 7.3, three classes of device may exist at each primary address on the bus. These are referred to as listeners, talkers and controllers.

Listeners

A listener can only receive data and commands from the bus; it cannot transmit them. A typical example of a listener is a printer which only receives data and control characters from other devices on the bus. There may be up to 14 active listeners present on the bus at the same time.

Talkers

Talkers are capable of transmitting data to other devices on the bus, but are incapable of receiving data or commands. Only one talker is allowed to be active at any one time.

Controllers

The controller supervises the transfer of data along the bus. This role is usually (but not always) performed by a PC equipped with a suitable IEEE-488 adaptor card. The controller can assign any device on the bus to act as a talker or listener. Many instruments are capable of acting as both a talker and a listener (and sometimes also as a controller). These devices are often dynamically switched (via commands sent from the current controller) between listener and talker modes. There may be more than one controller in the system but only one controller can be active at any time. The active controller can pass control to any other suitable device by issuing a Take Control (TCT) command. Before any data or messages can

be transferred over the bus, it is the responsibility of the active controller to initialize all other devices as either talkers or listeners.

Throughput

The IEEE-488 standard specifies that the maximum bit rate present on any one line of the bus must not exceed 1 Mbit/s. Some proprietary systems will allow significantly higher transfer rates. In practice, throughput will depend upon the performance of the IEEE-488 adaptor used, the PC's host bus (ISA, EISA, PCI, parallel port or RS-232 port) and driver software. In many cases, however, it is possible to attain data transfer rates of no more than about 250 KB/s using a standard IEEE-488 system. Transfer rates of a few hundred bytes per second are more typical when very slow devices are present on the bus.

The IEEE-488 handshake protocol guarantees that the overall speed of data transfer is determined by the *slowest* active listener present. This prevents data from being transferred too quickly for the listener to handle.

The handshaking protocol (discussed in more detail in the *Data transfer handshake* section later in this chapter) is fairly time consuming and can restrict throughput in some cases. National Instruments Corporation have developed a faster protocol, known as HS488. This is compatible with the standard IEEE-488.1 protocol, in so far as HS488 devices will employ the normal protocol to communicate with standard IEEE-488 devices. If all talkers and listeners on the bus are HS488 compliant, the faster protocol is automatically adopted. HS488 is implemented using special hardware and is software compatible with standard IEEE-488 systems. Slightly different cable-length restrictions apply, however. Throughput is dependent upon the host PC's bus and driver software, but 7.7 MB/s have been claimed for HS488 using a PCI bus-based adaptor under Windows NT. As HS488 is less widely used than the standard IEEE-488 protocol it will not be discussed further here.

The structure of the IEEE-488 bus

The bus consists of 16 signal lines together with a number of ground and shield wires. The IEEE-488 cable is usually terminated with a 24-pin Amphenol connector. The connector pin assignments are shown in Table 7.7.

Eight bidirectional data lines (DIO1–DIO8) are used for carrying data and command messages. The messages are transferred in accordance with a handshaking protocol implemented with the DAV,

Table 7.7 *IEEE-488 bus lines and connector pin assignment*

Pin	*Mnemonic*	*Name*	*Function*
1	DIO1	Bidirectional data bus lines	Transfer data or command codes.
2	DIO2		
3	DIO3		
4	DIO4		
13	DIO5		
14	DIO6		
15	DIO7		
16	DIO8		
6	DAV	Data valid	Asserted by talker to indicate bus holds valid data.
7	NRFD	Not ready for data	Asserted by listener to indicate that it cannot receive data.
8	NDAC	Not data accepted	Asserted by listener while reading data.
5	EOI	End or identify	Asserted by talker to identify the last byte of data in a block or message. Also used in parallel poll.
9	IFC	Interface clear	Asserted by controller to initialize all bus devices.
10	SRQ	Service request	Asserted by any device to request the attention of the controller.
11	ATN	Attention	Asserted by the controller to indicate that the data bus holds a command/address rather than data.
17	REN	Remote enable	Asserted by controller to disable any front panel controls.
18	DAV gnd		Ground.
19	NRFD gnd		Ground.
20	NDAC gnd		Ground.
21	IFC gnd		Ground.
22	SRQ gnd		Ground.
23	ATN gnd		Ground.
24	Logic gnd		Ground.
12	Shield		Shield.

NRFD and NDAC lines. In addition, five interface management lines (ATN, IFC, SRQ, REN and EOI) are used for carrying control and status information. All signals on the bus are active low – i.e. the lines are considered to be asserted (or active) when at a low logic level

(<0.4 V). All signal lines use TTL logic levels, although DAV, NRFD and NDAC employ open collector outputs. This allows them to be used in a wired-OR configuration so that any one of the bus devices can *independently* assert these lines. When unasserted, these lines lie at the logical high level of about 3.3 V.

Data transfer handshake

All message bytes are transferred from the talker to one or more listeners by means of a sequence of handshake signals. As mentioned previously, this process is designed to allow the slowest device on the bus to control the rate of data transfer. The handshaking sequence is illustrated in Figure 7.4 and is described below.

1. Each listener asserts the NRFD line while it is busy, only releasing it when it is ready to receive a message byte on the DIO lines from another device. Consequently, NRFD will go high (inactive) only when all active listeners are ready and have released NRFD. Each listener should also normally hold the NDAC line in an active state when ready for the next message byte.
2. Upon detecting that NRFD is inactive and NDAC is asserted, the talker places a message byte on the DIO lines.
3. The talker waits for 2 ms to allow the DIO lines to settle. It then asserts the DAV line to indicate that a valid message byte is present.

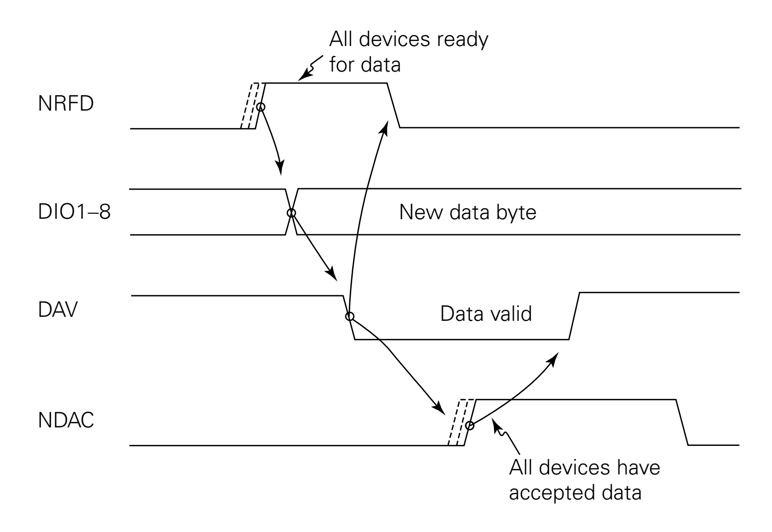


Figure 7.4 *IEEE-488 handshaking sequence*

4. The listeners, detecting that the DAV line has been asserted, begin to read the DIO lines. While performing this action, they each assert NRFD to indicate that they are busy.
5. Each listener acknowledges receipt of the message byte by releasing NDAC.
6. When all listeners have released NDAC, it goes high. This indicates to the talker that all of the listeners have accepted the message. The talker then completes the handshaking sequence by releasing the DAV line. At this point NRFD is still asserted, NDAC has been released, and the whole sequence may then be repeated in order to transfer the next byte.

Note that both NDAC and NRFD must be released by all listeners before they will go high. Each active listener releases these lines at its own rate and in this way the handshaking sequence is controlled by the slowest listener present on the bus. This prevents data from being transferred too quickly for the slowest listener to handle.

Interface management lines

The IEEE-488 bus possesses a number of lines for controlling devices on the bus, for issuing commands and for requesting service.

The IFC (Interface Clear) line may be asserted by the active controller to reset and initialize all bus devices. On receipt of this signal, the actions performed by each instrument connected to the bus will be device dependent. The IFC line is normally used by the controller at the beginning of a communications session to ensure that all devices are in a known default state. The controller asserts the ATN (Attention) line whenever it transmits a message that must be interpreted as a bus-management command, as opposed to a device-specific message or data (the differences between message types are described later in this chapter). When ATN is asserted, all devices on the bus will read any transmitted message byte, regardless of whether they have been configured as active listeners.

REN (Remote Enable) must be asserted to enable an instrument to be controlled by commands received over the bus. When REN is unasserted, the device can be controlled only via its front panel (if such facilities are available).

When a device on the bus requires attention from the controller – for example, if it has valid data available or if an error has occurred – it may assert the SRQ (Service Request) line. Upon detecting the SRQ signal, the controller will finish whatever task it is currently engaged in and then determine which device issued the request for service. Remember that the same SRQ line is shared between all bus devices, so when it is asserted, the controller only

knows that one (or possibly more than one) device requires attention. In order to detect which device issued the SRQ the controller initiates either a serial or parallel poll (see the following section). Each device responds to the poll command by issuing status information which informs the controller whether it requires service. The controller then services the appropriate device(s) by, for example, reading any available data.

Finally, the EOI (End or Interrupt) line is asserted by the active talker during transmission of the last byte of a multi-byte message. This provides a convenient means of identifying the end of a message or block of data. The EOI line also has an alternative use. It may be asserted by the active controller in conjunction with ATN in order to initiate a parallel poll as described in the following section.

Polling

The IEEE-488 interface implements a polling facility that allows the active controller to determine the status of each device on the bus. This is used, after the controller has received a Service Request (SRQ) signal, to determine which device needs attention. Two types of polling may be performed: serial or parallel.

A serial poll is enabled by issuing a universal SPE command (see the following section). This enables all devices on the bus in preparation for a serial poll. The controller then addresses each device, in turn, to talk by transmitting a TAG command. The device responds by transmitting a single status byte on the data bus. Bit 7 of the status byte is set if the addressed device is requesting service. The remaining bits carry device-dependent status information. When the serial poll has been completed, the controller usually issues the universal SPD (Serial Poll Disable) command so that normal bus operation can be resumed.

A parallel poll provides a faster alternative to the serial poll. This allows the controller to poll up to eight separate devices in one simple bus transaction. The devices participating in a parallel poll each transmit a status bit on one of the eight data lines. The bit allocations used by each device must previously have been programmed by means of the PPC (Parallel Poll Configure) command. The PPC command is first transmitted by the controller to a specific device. This is followed by a supplementary command byte, which assigns one of the eight data lines to the device for use in the subsequent parallel poll. The three low order bits of the supplementary byte contain the binary-coded ordinal index of the data line to be used. Note that the index runs from 0 (000b) for DIO1 to 7 (111b) for DIO8. Bit 3 of the supplementary byte indicates the polarity of the

device's status bit that is needed to request service: if bit 3 is high, the status bit must also be high during the poll in order to request service.

After all devices have been suitably configured, the controller is able to initiate a parallel poll at any appropriate time by simultaneously asserting the EOI and ATN lines. The devices on the bus respond by asserting (or unasserting) the appropriate data lines, indicating to the controller which devices require service. The universal PPU (Parallel Poll Unconfigure) command may be issued by the controller to disable the parallel poll facility.

Messages

So far we have referred only to messages being transmitted over the IEEE-488 bus. In fact, these messages can each be one of two types: data messages or bus-management commands.

Data messages

Data messages can represent just about anything that makes sense to a specific device. They can be pure data (e.g. the result of a measurement) or they may be device-specific commands. The form of a data message is purely device specific and is not defined by IEEE-488.1. Although some aspects of data messages are standardized in IEEE-488.2, many instruments employ completely different command sets. In an attempt to overcome some of the difficulties inherent in developing multi-instrument applications, a consortium of prominent IEEE-488 equipment manufacturers proposed a standard command set for IEEE-instruments in the early 1990s. This is known as Standard Commands for Programmable Instruments or SCPI. It visualizes every instrument as a hierarchical group of functional blocks and provides standard commands to control each block. This additional degree of standardization has the potential to greatly simplify programming and interchanging of instruments. A description of SCPI is beyond the scope of this book. For details, the reader is referred to programming guides supplied with SCPI compliant instruments.

Bus-management commands

Bus-management commands are not device specific. They are an essential part of the IEEE-488 standard and all devices on the bus must respond to them. The active controller can transmit bus-management commands to any or all devices on the bus. During transmission, the normal handshake protocol is used, except that

Table 7.8 *The IEEE-488 bus-management command byte*

Bit	*Description*
4–0	If bits 6,5 = 00: bits 0–4 hold the code of the Universal or Addressed Command. If bits 6,5 ≠ 00: bits 0–4 hold a primary or secondary address.
6,5	Command type: 00 = Bus command (for sending both Universal and Addressed Commands). 01 = Listen Address Group (for commanding a specific device to listen). 10 = Talk Address Group (for commanding a specific device to talk). 11 = Secondary Command Group (for accessing sub-units in a device).
7	Unused.

the controller first asserts the ATN line. This causes the active talker to relinquish control of the DAV line. The controller then becomes the active talker and is able to transmit command bytes.

When ATN is asserted, all devices read the commands that are transmitted by the controller, and participate in the handshake sequence regardless of whether they are configured as listeners. When the ATN line is unasserted, only the devices previously configured as talkers and listeners take part in subsequent communications.

The bus-management commands transmitted by the controller each take the form of a single byte, as shown in Table 7.8. Bit 7 (i.e. DIO8) is unused and should be zero. Bits 5 and 6 indicate the command group (i.e. the type of command that is being sent) and the remaining bits are interpreted either as a command code or as a primary or secondary address.

Addressed Command Group (ACG)

The commands in this group affect only those devices that have previously been addressed to listen. Bits 0 to 4 of the command byte specify the type of Addressed Command as shown in Table 7.9.

Universal Command Group (UCG)

The Universal Commands affect all devices connected to the bus. Bits 0 to 4 of the command byte specify the type of Universal Command as shown in Table 7.10.

Listen Address Group (LAG)

This group contains two commands which may be used to activate or deactivate a device's listen mode. In both cases bit 5 of the command

Table 7.9 *Addressed command group*

Command byte	*Name*	*Description*
01h	GTL	Go to local. Causes the device to be programmed locally (i.e. via its front panel). The device must be addressed to listen using the LAG command (see Table 7.8) in order for it to exit local mode. This command cancels the Universal LLO command for the listening device.
04h	SDC	Selected Device Clear. Initializes the listening device and resets it to its default state. The action performed is device dependent.
05h	PPC	Parallel Poll Configure. Configures the device to respond to a parallel poll signal (EOI + ATN asserted).
08h	GET	Group Execute Trigger. Simultaneously configures all devices configured to listen. Used to synchronize a group of devices to perform some pre-programmed task.
09h	TCT	Take Control. Issued by the active controller to cause the recipient of the command to take control of the bus. The new controller then becomes the active controller.

Table 7.10 *Universal command group*

Command byte	*Name*	*Description*
11h	LLO	Local Lockout. Disables the local (front panel) controls of all bus devices.
14h	DCL	Device Clear. Resets all devices. The action performed will be device dependent.
15h	PPU	Parallel Poll Unconfigure. Removes the parallel poll configuration of each bus device and prevents the devices from participating in a parallel poll.
18h	SPE	Serial Poll Enable. Sets all devices to serial poll mode. In this mode, each device will return one status byte when it is addressed to talk.
19h	SPD	Serial Poll Disable. Disables serial poll mode.

byte is set to 1 and bits 0–4 contain a primary address. The LAG command configures a specific device as a listener. The primary address of the device that is to listen (coded in bits 0–4) may fall in the range 0 to 30. The address value of 31 (i.e. bits 0–4 all set to 1) is invalid in the LAG command. Address 31 is known as the ‘unlisten

address' and a Listen Address Group command byte containing the unlisten address (i.e. 00111111b) defines the UNL (unlisten) command. This is used to globally disable all listeners on the bus.

When a device detects a LAG command in which bits 0 to 4 match its own primary address, it becomes an active listener. Thereafter, it reads all data bytes transmitted on the bus until it detects a UNL command.

Talk Address Group (TAG)

The talk address group contains two commands, TAG and UNT (untalk), which are analogous to the LAG and UNL commands described above, except that the TAG and UNT commands control which bus device is configured to talk. Commands in this group are distinguished from other command groups by the states of bits 5 and 6, as indicated in Table 7.8.

Secondary Command Group (SCG)

The Secondary Commands work in a similar way to the LAG and TAG commands in so far as they control which sub-unit in a previously defined talker or listener is active (i.e. transmits or receives data). Bits 5 and 6 identify the command as belonging to the Secondary Command Group.

Typical command and data transfer sequences

A simple example follows which will illustrate the sequence of commands and bus signals required to configure the talker and listener devices on the bus. The current controller must issue the following commands:

1. Assert the ATN line to identify the following as commands.
2. Issue an UNL command to unlisten all devices.
3. Issue a TAG command (including the appropriate talk address) to specify one talker.
4. Issue one or more LAG commands to specify one or more listeners.
5. Unassert ATN.

Suppose we subsequently wish to select the measurement range of a digital voltmeter on the IEEE-488 bus. The appropriate message to select measuring range 2 may, for example, be 'R2'. Note that this message will be device specific and may vary between different voltmeters. In the case of a SCPI compliant instrument, an appropriate SCPI command sequence would be used instead. If the message has

to be sent to primary address 10, secondary address 5, the following sequence would then be used.

1. Assert the ATN line to identify the following as commands.
2. Issue a UNL command to unlisten all devices.
3. Issue a LAG 10 command to cause the voltmeter (primary address 10) to listen.
4. Issue a SCG 5 command to access secondary address 5.
5. Unassert ATN.
6. Transmit an 'R' character.
7. Transmit a '2' character. This may be followed by a CR, LF pair. The EOI line is asserted during transmission of the last character in the sequence.
8. Assert ATN.
9. Issue an UNL command to unlisten the voltmeter.
10. Unassert ATN.

It is not practicable to attempt to cover device-specific command sequences here. Please refer to manufacturer's manuals for detailed information on configuring and operating specific equipment.

Interfacing IEEE-488 devices to the PC

The PC is usually interfaced to the IEEE-488 bus by means of an ISA, EISA or PCI adaptor card, although parallel port and serial port adaptors are also available. Most of these devices are software compatible with the 'industry standard' National Instruments GPIB-PCII and GPIB-PCIIA cards. The latter is functionally identical to IBM's GPIB adaptor. These cards conform to the IEEE-488.1 standard, but enhanced cards, which support the additional functions specified by IEEE-488.2, are also available. Adaptor cards usually allow the PC to act as a talker, listener or controller and allow up to 14 bus devices to be interfaced to the PC. The throughput offered by these cards varies, but most permit data transfer rates of up to about 300 KB/s.

Some adaptor cards include firmware drivers contained in ROM. The services provided by these drivers can be accessed via an interrupt interface in much the same way as BIOS services are invoked. Most cards, however, are accompanied by disk-based software which can be used by an applications program to communicate with the various instruments on the bus. Software drivers tend to take two forms: object files which can be linked to user written programs; or operating system device drivers (e.g. installable DOS device drivers or kernel-mode drivers under Windows NT) which are usually loaded into memory when the PC is booted. Operating system device drivers

are usually accessed from an application program via a special HLL library file supplied by the driver's manufacturer. Some manufacturers also supply configuration, diagnostics and development utilities, often as an integral part of the driver's API.

Software drivers are controlled with a variety of commands. Some commands are roughly equivalent to the single-byte bus commands, while others initiate lengthy sequences of bus transactions. Higher level commands are usually also available. These facilitate, for example, on-board buffering of data, control of multiple devices, and sophisticated bus management. Such a command mix provides the optimum combination of power and flexibility and means that there is usually no need for the programmer to be concerned with manipulating the interface hardware directly. The form and syntax of the commands tends to vary between the drivers offered by different manufacturers, but most provide a broadly similar set of functions. Note, however, that IEEE-488.2 drivers will include an extended API in order to accommodate the additional functionality encompassed by this standard. It is advisable to carefully study the manuals accompanying your IEEE-488 driver for full programming details.

8 Serial communications

As we have seen in the previous chapter, parallel buses provide a simple means of transferring data rapidly between the PC and external test instrumentation. They do, however, suffer from a number of limitations. Foremost amongst these are the expense associated with using long runs of multi-core cable and indeed the inability of many parallel buses to transmit over distances of more than a few metres. Each parallel interface also requires at least eight line drivers for the data bus and often several more to accommodate the various control lines, further increasing the cost of parallel bus interfaces.

Serial buses, on the other hand, provide a relatively cheap method of communicating over long distances. In serial systems, the data is broken down into a series of bit patterns and transmitted one bit at a time over a single wire (or pair of wires). This not only reduces the number of bus drivers needed and minimizes cable costs, it also allows data to be transmitted over very much greater distances. The RS-422 serial interface standard, for example, permits communication over distances of 1200 m using relatively inexpensive twisted-pair cable.

Serial transmission is normally slower than parallel I/O (although some serial systems allow for very high bit rates). With one or two exceptions, typical maximum serial transmission rates are about 10 KB/s with the PC. This is often quite adequate in data-acquisition, automation and industrial control applications where a throughput of 1–2 KB/s is more typical.

This chapter discusses the basic principles of serial communication and describes common standards and techniques that can be used for linking PCs and data-acquisition equipment.

8.1 Some common terms

Before proceeding with a description of serial communication systems, it is useful to define a few common terms.

Simplex and duplex communications

The terminology used to describe communication traffic can be confusing, primarily because different definitions of the terms simplex and duplex are used in the USA and in Europe. Because a majority of DA&C hardware, software and related literature originates from the USA, we will use the American National Standards Institute (ANSI) definitions throughout this book. The European alternatives are noted in the following paragraph.

The simplest form of serial communication involves transmission in a single direction, such as from a PC to some form of actuator or remote display unit. Unidirectional communication is termed simplex communication. Systems which allow data to be transmitted in two directions (i.e. to be transmitted and received by the same device) may be full duplex or half duplex. Half duplex interfaces (also known as simplex interfaces in Europe) accommodate transmission and reception, but not both at the same time, while a full duplex (duplex in Europe) device may transmit and receive data simultaneously.

Synchronous transmission

Synchronous serial transmission is the most efficient method of transmitting large quantities of data along a serial communications link. In a synchronous system, the link carries timing information which is used to synchronize the operation of the transmitting and receiving elements. The widely used RS-232 standard includes a number of control lines for this purpose, although these are not normally used in PC-based RS-232 implementations.

Data is generally transmitted in blocks which also contain various flags and header information. The advantage of this technique is that separate serial frames and the associated start and stop bits (see the following section) are not required for each transmitted character. This minimizes the overall time taken to transmit each byte. Synchronous transmission is used mainly in telecommunication and mainframe computer systems. As it is rarely used for data acquisition, it will not be discussed further in this book.

Asynchronous transmission

Asynchronous serial transmission is of more relevance to PC-based data acquisition. In an asynchronous system, the transmitter and receiver are *not* synchronized and each character is transmitted along the serial link independently of the last. In this case the

receiver automatically detects the start of each character and it then assumes that all subsequent data (and control) bits which constitute the character will arrive at a predetermined rate.

Usually a start bit is transmitted first and this alerts the receiver to the beginning of each new character. A series of up to 8 data bits are then transmitted and these are followed by one or more stop bits which mark the end of the character. An optional parity bit, which provides a limited error checking facility, is also sometimes transmitted immediately before the stop bit(s).

Transmission rate

The rate at which information is carried along the serial bus is measured in bits per second (bps) or, alternatively, baud. There is an important difference between these two terms although in many systems they are equivalent and are used synonymously. Technically, the baud rate refers to the number of discrete signal events (i.e. signalling elements or potential number of logical state transitions) occurring per second. In almost all asynchronous systems (with the exception of modem to modem communications), the state of each bit is coded by only one discrete signal event and thus the baud rate is numerically equal to bps. An exception to this is Hewlett Packard's Interface Loop (HP-IL) system in which each bit is represented by three state changes (or two discrete states). In this case the baud rate is not equal to the number of bits per second. Serial transmission rates usually range from about 50 baud to 115 200 baud and above, but most PC data-acquisition and industrial communications systems use baud rates in the range 1200 to 38 400.

8.2 Introduction to asynchronous communication

Asynchronous communication techniques are popular for industrial communication and for interfacing the PC to remote data-logging systems. PCs are normally equipped with at least one RS-232 port, although they can accommodate two or sometimes four separate ports. A number of other adaptor cards can be added to the basic PC architecture in order to provide RS-422 or RS-485 compatible communications facilities. Each additional port employs the same (or functionally compatible) type of controller (UART) as the standard RS-232 port and thus appears to the driving software to be identical at the register level.

The serial character frame

All characters transmitted asynchronously are packaged into a serial *frame*. This includes a start bit, the data bits and one or more stop bits. Asynchronous serial data can be framed and transmitted over RS-232, RS-422 or RS-485 buses in a variety of ways. However, the same protocol is used in the vast majority of cases.

When the transmitter is idle, the transmission line is forced to a logical high (or marking) state. The start bit consists of a single bit period (the length of which is dependent upon the baud rate or bps) during which the transmission line is placed in the logical low (spacing) state. The receiver detects the high-to-low transition which marks the beginning of the start bit and then prepares to receive a stream of up to 8 further data bits and an optional parity bit. Within the 8 data bits, the least significant bit is sent first. The serial frame is terminated by one or more stop bits, each consisting of a single bit period during which the transmission line is held in the marking (high) state. Figure 8.1 illustrates the usual form of the serial character frame. In this example the value 45h (i.e. ASCII 'E' or 01000101b) is coded into a stream of 8 bits. This is preceded by the start bit (which is always low) and in this example followed by an odd parity bit and 2 stop bits.

The parity bit provides a limited error checking facility by indicating whether the total number of high data bits is odd or even. In an even parity system, the state of the parity bit transmitted within each serial frame is such that the number of high bits contained within the data-plus-parity bit pattern is even. If odd parity is selected, the converse is true. Thus if 1 data bit is incorrectly detected by the receiver (due to noise on the transmission line, for example), there will be a mismatch between the high bit count and the parity bit. The receiver will then be able to flag the received character as being

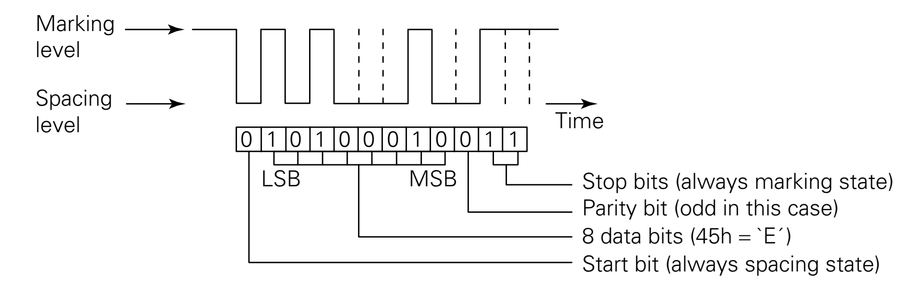


Figure 8.1 *The serial character frame*

corrupted. This technique does not, of course, allow more than one erroneous bit to be detected in each serial frame.

In order to discover the state of each bit, the receiver samples the transmission line at times corresponding to the centre of each bit. In fact, each bit is usually sampled more than once in order to enhance the system's noise immunity. The timing of each sample is performed relative to the beginning of the start bit. Both the transmitter and the receiver contain clocks, which are used to time the transmission and sampling of the bit stream. Because the start bit provides a means of synchronizing both devices, this method of communication is relatively insensitive to small inaccuracies in the timing elements. Timing variations of up to about 5 per cent can be accommodated in most systems.

Handshaking

It is obviously essential for the transmitting and receiving devices to agree when to allow data to be transferred. This requires some independent method of communication so that the transmitter does not place characters on the bus until the receiver is ready. Additional control lines are incorporated into some serial buses for this purpose. These enable the bus device to signal that it is ready to communicate and to request (and then to receive) clearance to transmit data. This technique is termed hardware handshaking.

A number of control lines are specified by the different serial communications standards (such RS-232 or RS-422), but within each standard, there is some variability as to which of the available control or handshaking lines are actually used. Some systems employ quite extensive handshaking, using three or four control lines, while others dispense with hardware handshaking completely. Hardware handshaking in RS-232 systems is discussed in the section *Control lines, handshaking and null modems* later in this chapter.

In cases where no hardware handshaking is used, other techniques must be employed. These can range from simple timing loops, which prevent devices from transmitting at certain pre-arranged times, to rules governing the type and length of messages that may be transmitted. Often one of the devices on the serial bus (usually the PC) is designated as a controller and only this device is allowed to initiate activity on the bus. The listening device (e.g. a remote data logger) might then be required to respond to commands from the controller within a predetermined time limit. Often the controller will transmit characters one at a time and wait for the listening device to respond by echoing the character back. This has the benefit of

simplifying error detection, although it does slow down the overall transmission rate.

Conventional software flow-control protocols allow the receiving device to control the rate of data flow by transmitting special control characters. When these are detected by the transmitter, it temporarily suspends transmission. Flow control protocols usually use the XON (DC1) and XOFF (DC3) ASCII control characters to enable and disable transmission, although other characters are sometimes employed for this purpose.

Timing, echoing and XON/XOFF flow-control techniques are usually quite simple to implement in PC-based data-acquisition systems. Because no control lines are needed, inexpensive two- or three-core cable can be used. This tends to make software flow control somewhat cheaper to implement than hardware handshaking, particularly where long cable runs are required.

The UART

The PC's asynchronous serial communications interface is controlled by a device known as a UART (standing for Universal Asynchronous Receiver/Transmitter). This component usually takes the form of a single IC, although a few data-acquisition and intelligent signal-conditioning products simulate the actions of a UART in software. The UART automatically converts all data which the software writes to its transmitter register into serial format and then adds the necessary start, stop and parity bits. The serial bit pattern is transmitted at a frequency consistent with an agreed and preprogrammed baud rate. A UART in the receiving device detects each bit in the serial frame, strips out the start, stop and parity bits and converts the data back into a parallel (byte) format which can be read by the receiving software. The receiving UART usually performs some limited error checking (e.g. for parity and errors in the composition of the serial frame) and sets status and error flags which may be read by the receiver's software.

UARTs usually also possess several digital inputs and outputs. These are used primarily to drive and sense the hardware handshaking lines although, as we will see later, they sometimes serve other purposes. The digital I/O lines are generally accessed by the software via the UART's registers.

The UART may also provide interrupt facilities. These allow the communications port to interrupt the current program in order for the processor to perform an urgent task such as reading the next received character. Interrupt facilities can, in many instances, reduce

the software overhead by allowing the transmitting or receiving process to continue with other tasks until the UART requires service.

The various UARTs used on the PC are discussed in more detail later in this chapter.

Serial protocols

The term 'protocol' refers to the set of rules that specify how data is to be encoded as a serial bit stream, transferred along the communications link and then interpreted by the receiver.

Handshaking and the serial frame together form what might be termed the low level or byte-transfer protocol. This specifies how communication is to be established and how individual bytes are encoded into a serial bit pattern.

A higher level protocol defines the format of data as well as the timing and the nature of messages that pass between the various devices on the bus. With a few exceptions, there is very little standardization between serial-bus DA&C devices. Most devices use a command protocol based on short strings of characters. Because of the variety of different command sets in use, it is inappropriate to attempt to cover them here other than to mention some common character encoding schemes. The most widespread of these is the ASCII scheme which is described in Appendix B. This assigns each of 128 characters to a unique 7-bit binary number. The first 32 of these characters are designated as control characters and are used for actions such as software flow control. The XON, XOFF, SOH, ENQ, ACK, NAK and EOT characters referred to below are all ASCII control characters. Several other character coding schemes may be used and these are discussed in Appendix B.

Networks of serial devices (see the *Serial network and bus structure* section later in this chapter) will usually be designed to operate in the absence of any synchronization mechanism – i.e. using a so-called asynchronous protocol. In such a system, one device is designated as a bus controller. Typically, when power is first applied, all devices on the bus will enter their receive mode. The controlling device (usually the PC) will then initiate each bus transaction by sending commands to one or more devices, which will respond by transmitting a block of data or some form of acknowledgement back to the PC. Timeouts are usually applied in order to guarantee that the network returns to a known state in the event of a communication error. Error checking schemes may also be incorporated into the protocol.

There are several ways in which data can be packaged and transmitted. The most efficient protocols allow data to be buffered and transmitted as one large block. Block transmission techniques, which

are normally referred to as file transfer protocols, usually require a header block to be transmitted before any data. The header might contain information to identify the data block being sent, the number of bytes in the block, and special control characters (e.g. ASCII 01h, SOH) to mark the start of each header. The header can also facilitate implementation of error detection schemes by allowing checksums to be transmitted along with the data block. The data encapsulated in each block might represent text (using, for example, the ASCII encoding scheme) or it might represent a series of binary codes – ADC readings, for example.

Protocols such as XMODEM or KERMIT are commonly used for transferring files between computers. These are generally less useful in data-acquisition applications although similar, but less complex, systems are sometimes employed for downloading readings from a remote data logger.

Block transfer usually requires some form of software handshaking in order to allow the receiver to control the rate of data flow. The XON/XOFF protocol has already been discussed, but other techniques employing, for example, ENQ/ACK or ACK/EOT can be used.

The ENQ/ACK protocol allows the transmitting device to poll the receiver in order to determine whether it is ready to receive a block of data. The transmitter first sends the ENQ character and waits until it receives an ACK character back from the receiver before it starts transmitting the block of data. When the transmission is complete, the transmitter continues polling the receiving device by sending ENQ characters.

In the ACK/EOT protocol, the receiver initiates transmission by sending an ACK character to the transmitter which, in turn, transmits a block of data. When it has finished, the transmitter then sends an EOT character to mark the end of transmission. The XMODEM protocol employs a similar technique, but uses ACK only to request the next data block in a sequence. The NAK character is sent instead to initiate transmission or to request retransmission of the previous block.

The reader is referred to Stallings (1997) for more on handshaking, protocols and error detection.

8.3 Data acquisition via a serial link

Serial interfaces are often used to communicate with remote data-logging stations or signal conditioning modules. The simplest serial data-acquisition and control devices possess no on-board

processing capability and these usually operate as basic parallel-to-serial converters, allowing digital I/O lines and ADCs to be controlled or sensed via the serial port.

More typical data-logging modules incorporate their own processing units that can be configured or programmed via the PCs serial port. Often, these devices can acquire and log data independently of the host PC. Many can also perform basic control operations and execute simple data-reduction algorithms which obviate the need to transmit large quantities of data back to the PC. Indeed, some data-logging stations can operate independently in the field for many days or weeks and can then periodically download the acquired data to a portable PC for permanent storage and analysis.

Intelligent data-acquisition units can usually be configured to automatically scale and linearize acquired data. Calibration scaling factors and linearizing polynomials (see Chapter 9) can be downloaded to the unit prior to the data-gathering period. By issuing suitable commands, the PC can cause the data-acquisition unit to perform operations such as correcting for zero-drift, setting the sampling rate or configuring comparators. Acquired data might be transmitted back to the PC in text or 16-bit binary-word format. The latter is suitable for transmission of unscaled ADC readings. However, text transmissions are usually used for scaled data which has to be represented in floating-point format. (One may, of course, encode floating-point scaled data in 48-bit, 64-bit or 80-bit binary format for transmission, but this is rarely done in data-acquisition applications.)

Apart from the independence and parallelism which intelligent data-acquisition units offer, one of their main advantages is that they are often small, portable devices and can usually be sited in quite remote and inhospitable environments. This type of installation requires a robust, long-distance communications link. Such a link can be established using one of the serial interface standards such as RS-422. In long-distance communications systems, the cost of cabling can be a significant consideration and in order to minimize this, handshaking and other control lines are often dispensed with. Communication then takes place using only single or double twisted-pair cables. Data-acquisition systems of this type tend to employ software flow-control and/or character-echoing techniques instead of a hardware handshaking protocol.

Serial network and bus structure

A number of different interconnection schemes can be used in serial data-acquisition systems. Several examples are shown in Figure 8.2.

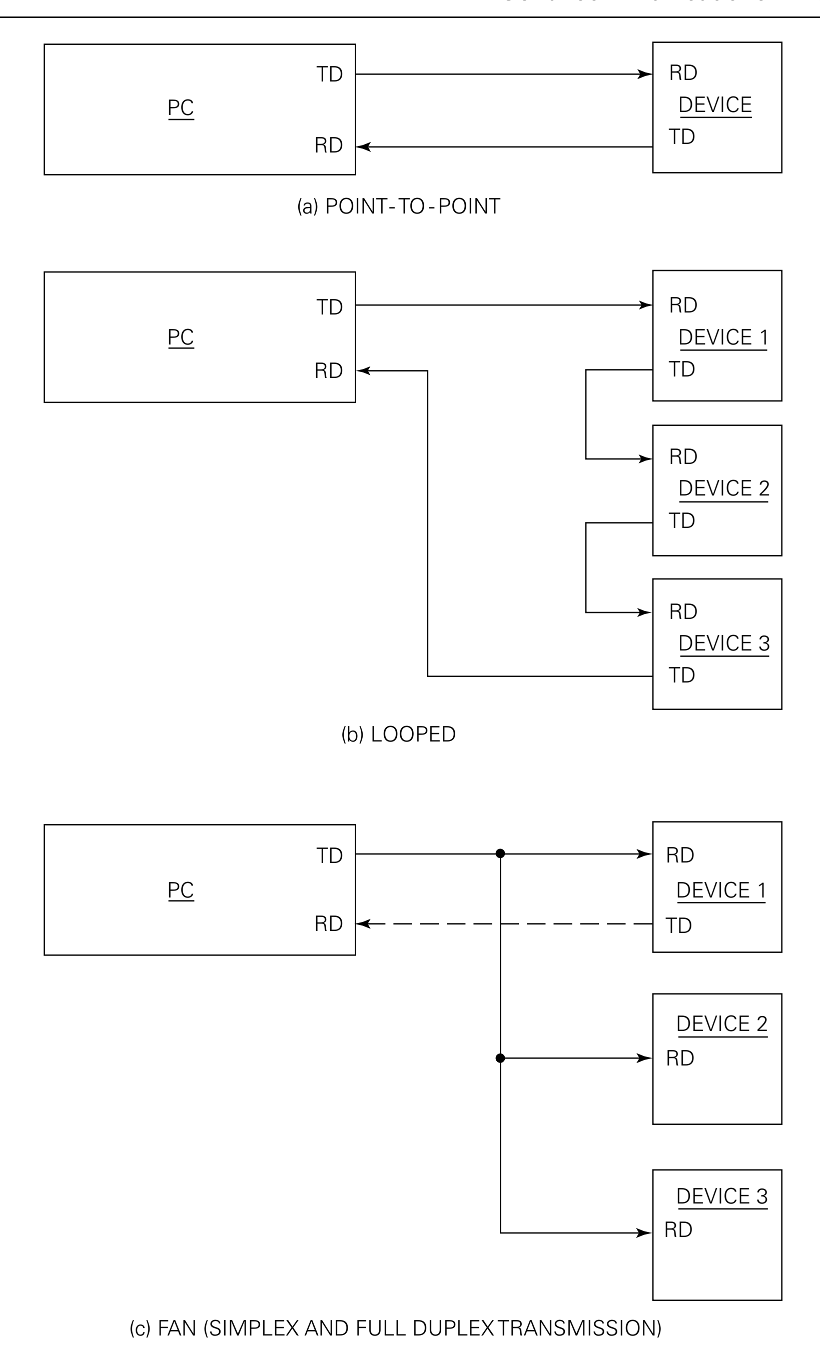


Figure 8.2 *Serial network topologies*

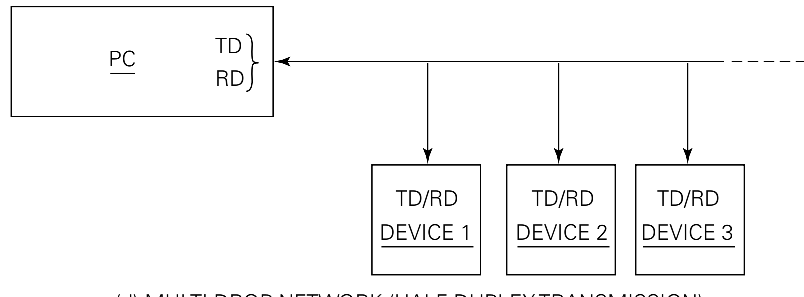


Figure 8.2 *(continued)*

Each interconnection line in this figure represents a single-ended electrical connection in the case of RS-232, or a differential connection in the case of RS-422/485 interfaces. The simplest scheme is the linear, point-to-point arrangement shown in Figure 8.2(a). This is the ideal arrangement where only one device has to be connected to the PC. Simplex, half duplex and full duplex systems can be supported using this structure. All of these can be implemented using the RS-232 or RS-422 standards. Point-to-point systems can be extended to form a loop structure in which each device on the network receives data or a command from an adjacent device and then relays it to the next device in the loop as shown in Figure 8.2(b). The data continues to be passed around the loop until it returns to the device that originally issued it. As well as making for an orderly communication protocol, this also allows the originator of the data to check that the echoed character matches that originally transmitted and thus to ensure complete data integrity. However, the repeated relaying of data does tend to slow the whole operation, particularly when low baud rates or slow devices are present on the loop.

Figure 8.2(c) shows an alternative serial network topology that can be used with interfaces conforming to the RS-422 standard. This allows the PC to transmit data to a number of separate devices, but only one of these devices can transmit data back to the host PC.

Finally, we have the so-called multi-drop, or bus, arrangement shown in Figure 8.2(d). This allows several transmitting and receiving devices to be connected to the same bus without the need to relay data from one device to the next. The multi-drop bus topology can be implemented with devices conforming to the RS-485 standard. Because it allows multiple transmitters and receivers to reside on the same bus, this arrangement can accommodate only simplex or half duplex operation. It is, however, very useful for interconnecting distributed signal-conditioning modules, such as might be employed

to monitor movement or loads at different points on a bridge, for example.

Speed and transmission distance

The maximum practicable rate of transmission along a modemless serial communications link varies with the total distance between the transmitter and receiver. The resistance and capacitance inherent in long cables tends to round off the sharp transitions present in digital signals. The effect of this rounding is most apparent when short duration pulses have to be detected (i.e. at high baud rates) and there is consequently a reciprocal relationship between the maximum baud rate and the total transmission distance. Note that while a system might operate satisfactorily with cables of a certain length it is always good practice to use the shortest practicable cable runs. Marnham (1994) discusses cable length calculations in some detail.

The RS-232 standard is capable of transmitting data over distances of up to 15 m, and speeds of up to 20 Kbps can be employed. Although this is often adequate for use within the limited confines of a laboratory, RS-232 is not a suitable solution for communicating with remote and inaccessible devices.

The RS-422 and RS-485 standards accommodate total transmission distances of up to 100 m at 1 Mbps using suitable twisted-pair cable. The maximum transmission rate is also much greater, being up to 10 Mbps. Typically this transmission speed is used over distances of less than 15 m. Such high rates cannot normally be achieved with the PC and an upper limit of 115 200 baud is imposed by the baud rate generator circuitry present in the PC's UART. Because of the lower transmission speeds possible with the PC, the recommended maximum cable lengths can be exceeded in some situations without introducing an unacceptable level of communication errors. If suitable drivers and/or cables are employed in RS-422/485 systems it is possible to extend transmission distances up to around 1200 m.

The highest transmission rates normally employed for long-distance (i.e. up to 1200 m) communications via RS-422 and RS-485 interfaces are about 19 200 to 38 400 baud. If lower baud rates are used, these interfaces will often tolerate even longer cables. RS-422/485 transmitters are available for transmission up to 11 000 m (7 miles) at 1200 baud.

Special signal converters are also available to extend the transmission range of standard RS-232 interfaces. These use fibre optic or current loop techniques. The latter will generally accommodate

data rates of 9600 baud or greater and will transmit over distances of several kilometres (typically 1000 to 8500 m).

8.4 Serial interface standards

We have already mentioned the standards, such as RS-232, developed by the Electronic Industries Association (EIA). This section outlines some important characteristics of these standards.

The RS-232 standard

The RS-232 standard was developed in the 1960s for transferring data between computers and peripheral devices (teletypes and printers), and for intercomputer communications. The RS-232C revision is most widely complied with, although it was superseded in 1987 by RS-232D and then in 1991 by revision E. From the PC programmer's perspective, however, the differences between the various revisions are of little significance and so this standard will be referred to simply as RS-232 in the remainder of this book. Additional information can be found in the texts by Putman (1987), Maine (1986), Marnham (1994) and Tooley (1992).

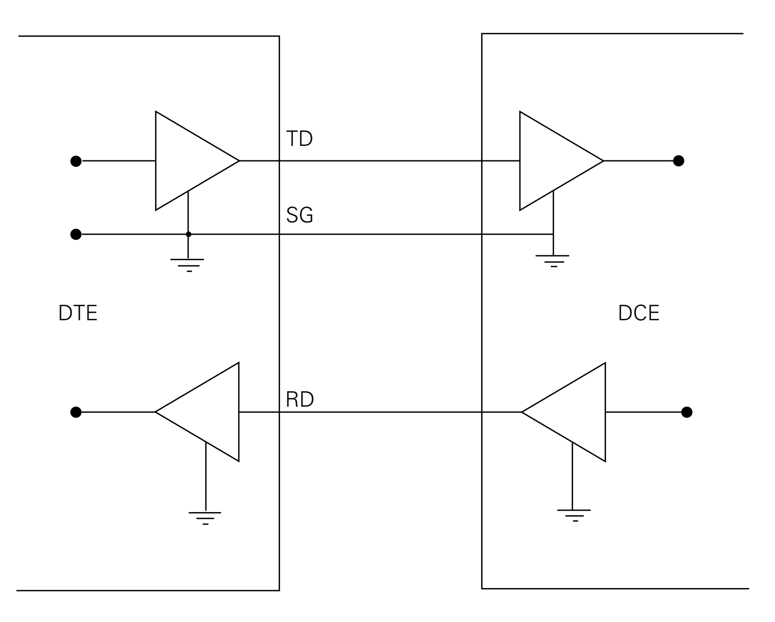


Figure 8.3 *Single-ended serial transmission over an RS-232 interface*

This standard is used for interfacing computers to modems for long-distance communication via the telephone network. RS-232 was originally designed with this application in mind and much of the terminology used (e.g. names of control signals etc.) reflects this.

RS-232 specifies a single-ended transmission system in which transmitted signals and received signals are each carried on a single wire. The voltage on each wire is measured with reference to a common signal ground as indicated in Figure 8.3. As mentioned

Table 8.1 *RS-232D connector pin assignments*

25 way	*9 way*	*Signal/circuit mnemonic*	*I/O relative to DTE*	*Full name*
1	–	FG/AA	–	Frame ground
2	3	TD/BA	Out	Transmit data
3	2	RD/BB	In	Received data
4	7	RTS/CA	Out	Request to send
5	8	CTS/CB	In	Clear to send
6	6	DSR/CC	In	Data set ready
7	5	SG/AB	–	Signal ground
8	1	DCD/CF	In	Data carrier detect/received line signal detect
9	–	–	–	Reserved/Testing
10	–	–	–	Reserved/Testing
11	–	–	–	Unassigned
12	–	SCF	In	Secondary CF (DCD)
13	–	SCB	In	Secondary CB (CTS)
14	–	SBA	Out	Secondary BA (TD)
15	–	DB	In	Transmitter signal element timing
16	–	SBB	In	Secondary BB (RD)
17	–	DD	In	Receiver signal element timing
18	–	LL	Out	Local loop-back signal
19	–	SCA	Out	Secondary CA (RTS)
20	4	DTR/CD	Out	Data terminal ready
21	–	CG	In	Signal quality detector
22	9	RI/CE	In	Ring indicator
23	–	CI/CH	In/Out	Data signal rate selector
24	–	DA	Out	Transmitter signal element timing
25	–	–	–	Unassigned/Testing

in the previous section, this single-ended operation restricts the maximum baud rate and transmission distance.

The RS-232 specification allows for only one transmitter and one receiver to be present on each signal line and this limits the topology to a linear point-to-point arrangement or a loop structure (see Figure 8.2). Full duplex, half duplex and simplex transmission modes can be used.

Connector pin assignments

RS-232 specifies a 25-way D-type connector with the pin assignments listed in Table 8.1. Two separate serial communications channels are supported by the standard, but only one of these – the Primary channel – is used on the PC's serial ports. The slower Secondary RS-232 channel is not available on the PC and this is reflected in the connector pin usage. All 25 pins are defined by the RS-232 standard, although only nine of these are in common use. Most modern PCs make only these nine signals available via a 9-way D-type connector (also listed in the table). Some IBM PC/XT or AT clones possess a 25-way connector, but with only the nine commonly used pins connected.

The RS-232 signals can be divided into four classes: data, control, timing and ground. The timing signals are defined for use in synchronous communication systems. Because the PC's serial ports support only asynchronous communication, these timing signals are not present.

Voltage levels

RS-232 defines the digital logic levels shown in Table 8.2. These levels are used on both the data and control lines.

This definition of logical states is used to represent the bit pattern within each serial frame. A logic 1 level, equivalent to a negative voltage, represents a high (1) data bit. The control lines are, however, generally active (i.e. on or asserted) when at logic zero (i.e. a positive voltage).

DTE and DCE

When considering the RS-232 interface you should remember that it was originally designed for connecting a computer terminal to a

Table 8.2 *RS-232 voltage and logic levels*

Logic level	*Voltage*	*Data line state*
0	+3 V to +25 V	Space
1	−3 V to −25 V	Mark

modem in order to facilitate communication with a remote (usually mainframe) computer. The modem performed the task of communicating over a long distance (i.e. telephone) link with a remote modem. At one end of the link, the local modem was connected to a computer terminal using an RS-232 interface and, at the other end, the remote modem was coupled to the remote computer, also by means of an RS-232 standard interface.

For this reason, RS-232 systems use terminology relevant to this mode of communication. Data Terminal Equipment (DTE) refers to those elements of the system that reside at the termini of the communications link. In the terminal-to-computer example, both the terminal itself and the remote computer would be classed as DTE. The modems, which established the long-distance link, are classed as Data Communications Equipment (DCE).

Of course, in the context of PC-based data-acquisition systems, the computer (i.e. PC) and the terminal are one and the same, and the RS-232 communications link is established between the PC and a device such as a data-logging unit, without the aid of a modem. In this case both the PC and the data logger are classed as DTE. No DCE (modem) is used.

Control lines, handshaking and null modems

The handshaking protocols used in RS-232 systems stem from the standard's original function as a way of connecting DTE and DCE. Table 8.3 lists the common handshake lines available on the standard RS-232 connector.

These lines are also present on the PC's 9-way connectors. The table provides a summary of the original usage of the various control lines, but this should be treated as only a very rough guide. In PC-based data-acquisition (and other) systems, the handshaking lines are actually used in a variety of different ways. In some cases, most or all of the lines are used; in others, only one or perhaps two of the available signals are needed. A number of systems dispense with hardware handshaking altogether. The timing of the handshaking signals also varies to some extent.

Some common handshaking sequences are listed below. Note that the RI and DCD inputs to the DTE are not checked in these examples, although they may be used in some applications. The RI signal indicates that the DCE has detected a ringing signal from the remote equipment. DCD is generally asserted when the DCE (modem) detects a carrier signal from the remote equipment. In applications where the DCE is actually a data logger or similar, the DCD line may be asserted when the logger is switched on and

Table 8.3 *Common RS-232 handshaking lines*

Mnemonic	*I/O relative to DTE*	*Usual use*
DTR	Out	Indicates that the DTE is ready and causes the modem to establish the long-distance (telephone) link.
DSR	In	Indicates that the modem is ready, but does not necessarily indicate that the remote communications link has been established.
DCD	In	Indicates that the local modem has detected the data carrier signal from the remote modem and that the remote communications link has been established.
RTS	Out	Indicates that the DTE is ready to transmit.
CTS	In	The modem asserts this line, in response to DTR and RTS, when it is ready to allow the DTE to transmit. RTS should go inactive after CTS has been asserted. RTS should not then be activated again until CTS is unasserted.
RI	In	Indicates that the local modem is receiving a ringing signal from a remote device. This is normally used by communications software to answer an incoming call.

functioning correctly. More commonly in this type of application, however, neither DCD or RI are used.

Transmission

1. DTE asserts DTR to indicate that it is ready to communicate.
2. DTE waits for the DCE to respond. DCE responds by asserting DSR. The assertion of DSR generally means that the DCE is ready; it does not necessarily mean that the DCE has established a communications link to the remote equipment. If the DSR line is not asserted within a predetermined timeout period (usually about 2 to 10 ms), the DTE assumes communication with the DCE cannot be established and times out.
3. DTE asserts RTS to request permission to transmit.
4. DTE waits for the DCE to assert CTS. If this line is not asserted within a predetermined timeout period (usually about 2 to 10 ms), the DTE assumes communication cannot be established and times out.

5. If a timeout did not occur, the DTE transmits the data. Either single characters or a block of several characters may be transmitted.
6. The DTE deactivates RTS at the end of transmission. Once RTS is deactivated, it should not be reasserted until after the DCE has deactivated CTS. DTR may remain active if the DTE wishes to stay on line.

Note that, in some systems, the transmission handshake is implemented using only the RTS/CTS handshake and the DTR and DSR lines are unused. The above transmission sequence is illustrated diagrammatically in Figure 8.4. The circled numbers in the figure refer to the steps in the foregoing sequence.

Reception

1. DTE asserts DTR to indicate that it is ready to communicate.
2. DTE waits for the DCE to respond. DCE responds by asserting DSR. If this line is not asserted within a predetermined timeout period (usually about 2 to 10 ms), the DTE assumes communication cannot be established and times out.
3. If a timeout did not occur, the DTE waits to receive data from the DCE. Either single characters or a block of several characters may be transmitted.
4. The DTE may deactivate DTR at any time to suspend the DCE's transmission.

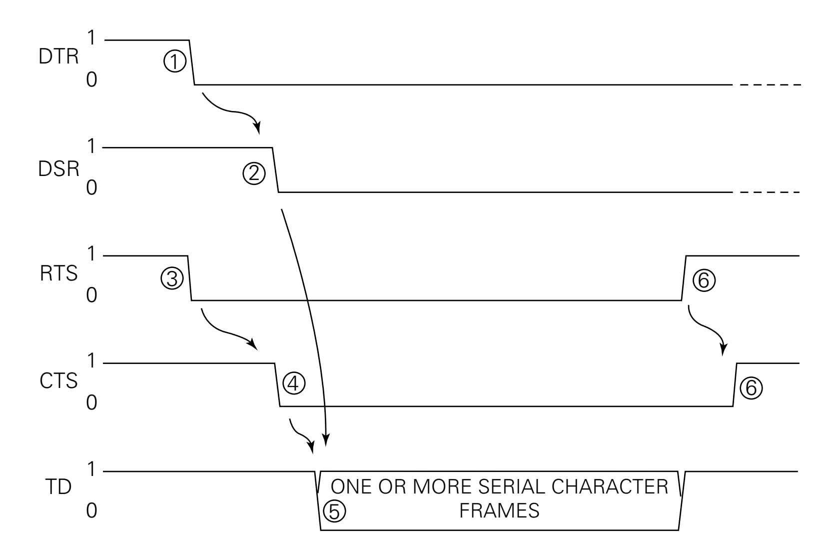


Figure 8.4 *Typical handshaking sequence used during serial transmission*

Null modems

In a PC-based data-acquisition system, there is generally no modem (DCE), and the PC is usually connected directly to a data logger or signal-conditioning module. Both the PC and data logger (etc.) are classed as data terminal equipment (DTE). In this case it is often necessary to make it appear to each DTE element that the handshaking signals have originated from a modem. To this end, special cables or adaptors, known as null modems, can be used. These employ crossed wiring which causes, for example, the TD pin of one terminal to be connected to the RD pin of the other (and vice versa). The handshaking lines are also crossed and/or looped back so as to emulate the signals that would otherwise have been provided by a modem. The exact design of these adaptors depends upon the requirements of each application and there is some variability in the wiring schemes used.

Figure 8.5 illustrates the connections employed in a variety of common null modem adaptors. These fall into two categories. Loop-back adaptors feed the control outputs (DTR and RTS) back to the input lines (DSR, DCD and CTS) of the same device. These connections do not provide any real handshaking facilities: they are merely used to circumvent any handshake requirements that

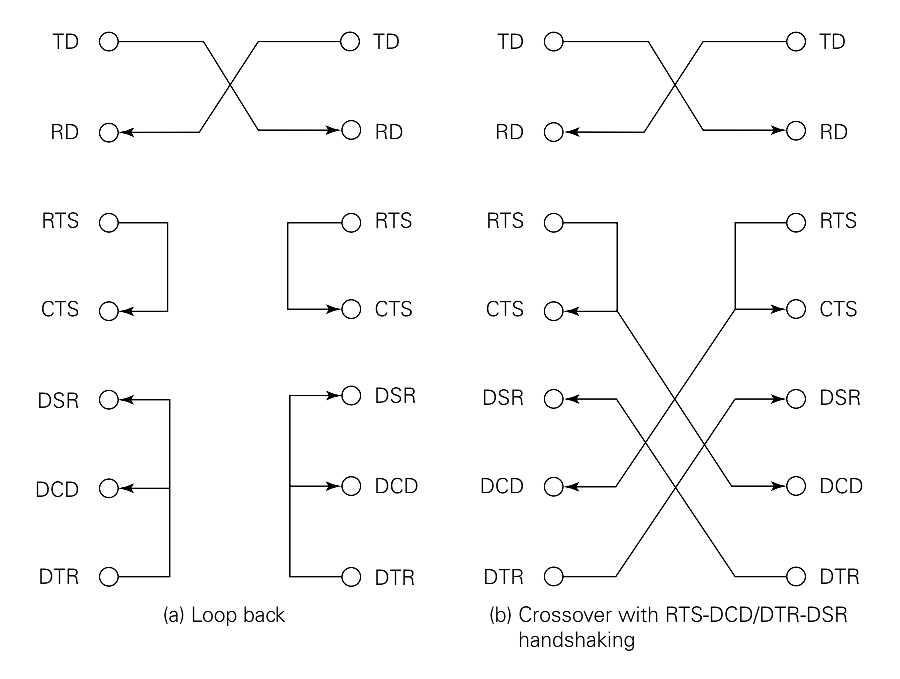


Figure 8.5 *Some common null modem connections*

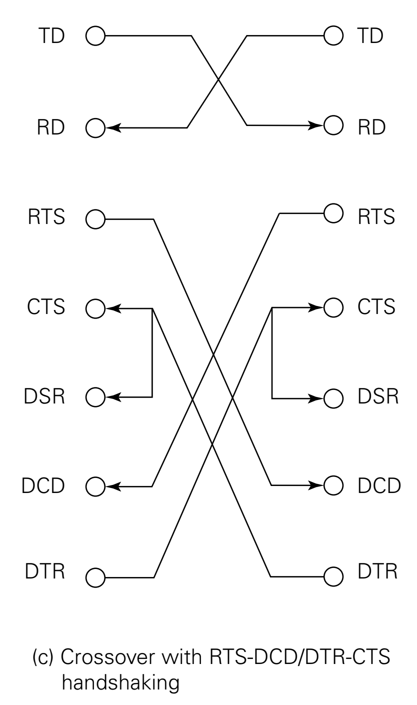


Figure 8.5 *(continued)*

may have been imposed by the DTE's communications software. The second class of null modems does implement some degree of handshaking between the two DTEs. In this case, crossed wiring is used to simulate the effect of communicating with a local modem.

There are several other types of null modem and crossed-wire adaptors. Some are required specifically for applications such as interfacing to a printer via the serial port. Devices, known as breakout boxes, are available which allow the various interconnections to be made and easily modified. These are ideal for experimentation in order to establish the correct null modem connections for use with an unfamiliar system.

The RS-422 standard

This standard is used widely in industry for communicating over longer distances than is normally practicable with RS-232. It was revised in 1994 and this revision is known as RS-422B (or EIA/TIA-422-B). Unlike the RS-232 standard, in which the signal voltages are all measured with reference to a common ground wire, RS-422 systems employ balanced differential transmission. In this mode,

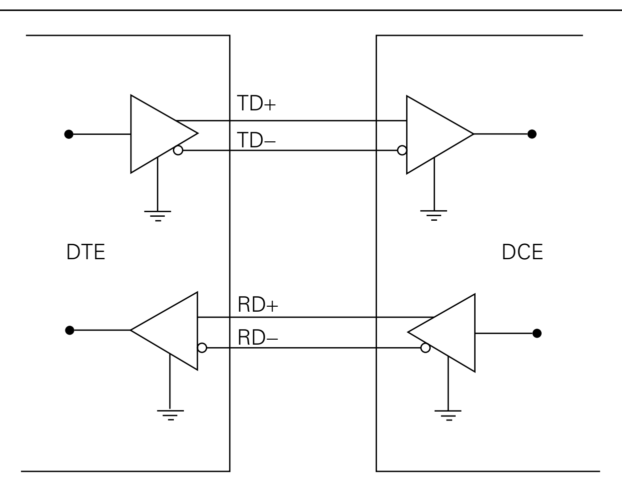


Figure 8.6 *Balanced differential transmission over an RS-422 interface*

signals are transmitted by means of pairs of wires which are labelled TD+ and TD− for the transmission circuit or RD+ and RD− at the receiver (also sometimes referred to as TD and TD Common or RD and RD Common). This transmission mode is illustrated in Figure 8.6 and should be compared with the single-ended mode employed by RS-232 (Figure 8.3).

Differential transmission permits some RS-422 compatible line drivers to achieve data rates of up to 10 Mbps over distances of around 300 m, although many standard RS-422 devices are capable of transmitting up to only 12–15 m at this speed. However, this is largely academic when using the PC as the standard 16450 UART can transmit at up to only 115 200 baud, and the maximum practicable transmission rates are often considerably lower. The transmitter and receiver can be separated by up to 1200 m provided that lower transmission rates (i.e. no more than 19 200 to 56 000 baud) and suitable twisted-pair cables and line drivers are employed. As noted earlier, RS-422 compatible transmitters are available for communicating over distances of up to 11 000 m (7 miles) at 1200 baud. Note, however, that the maximum recommended cable lengths tend to vary somewhat between different proprietary RS-422 compatible systems and you are advised to consult the manufacturer of your equipment for precise details.

Because separate TD and RD circuits are used, RS-422 is suitable for full duplex communication. RS-422 can also accommodate up to ten receivers on the same bus although, like RS-232, only one transmitter can be present. This allows a point-to-point, looped or fan topology to be employed.

The connector pin assignments used on industrial RS-422 devices tend to vary somewhat, although most interfaces and converters employ a 9-way male D-type connector incorporating pairs of pins for the TD, RD, RTS and CTS signals together with a single signal-ground pin. Only RTS/CTS handshaking is normally possible, because lines such as DTR, DSR, DCD are normally not required for RS-422 communications and are not present on the RS-422 connector.

The RS-485 standard

The RS-485 standard (introduced in 1983) can be considered an adaptation of RS-422 which allows many drivers and receivers to be present on the same bus (although only one driver may be active at any time). RS-485 employs balanced differential signal lines, much like RS-422. Full duplex implementations are possible using a point-to-point topology and separate twisted-pair conductors for the receive and transmit signals. In addition, RS-485 facilitates construction of multi-drop networks. This arrangement uses the same pair of wires for both transmission and reception of data. Although this helps to reduce cabling costs, it precludes full duplex operation. Figure 8.7 illustrates the structure of the half duplex RS-485 bus.

Notice that both the transmitter and receiver are connected to the same pair of wires. The transmitter and receiver are collectively known as a transceiver. Each device controls whether the transmitting or receiving element of the transceiver is active by means of the digital TE (Transmit Enable) line. This line may be driven either by software or by circuitry which senses when the device begins to transmit.

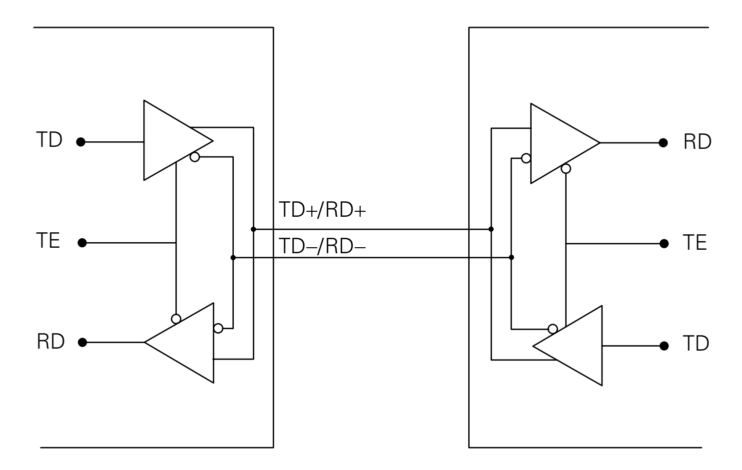


Figure 8.7 *Half duplex transmission over a balanced differential RS-485 bus*

The maximum permissible data transfer rates and cable lengths are similar to those described for the RS-422 standard. RS-485 will, however, support up to 32 drivers and 32 receivers on the same bus, although only one driver may be allowed to transmit at any time.

Like RS-422, the connector pin assignments used on many industrial RS-485 devices also tend to vary. In fact, the connector type and pin-out are not defined under the standard (Marnham, 1994). Some interfaces and converters employ a 9-way male D-type connector incorporating a pair of pins for the TD/RD signal together with ground and +5 V connections. Some devices offer both RS-422 and RS-485 operation and provide a dual-purpose connector. In RS-422 mode, the connector provides the normal RS-422 pins described in the previous section. In RS-485 mode, the RD pair of inputs is unused – the TD pins may then be used for both transmission and reception.

The half duplex nature of the RS-485 bus, together with the lack of handshaking in many implementations can make the design of protocols and message timing more complicated than with RS-232 or RS-422, and can place an additional burden on the software designer. In addition, as noted previously, it is necessary for each device on the bus to independently enable and disable its transmitter and receiver by controlling the state of the TE line. (Note that the TE line is not part of the RS-485 bus: it simply controls the direction of data flow through the transceiver.)

Some RS-232 to RS-485 converters that connect directly into the PC's RS-232 serial ports use one of the handshaking lines (e.g. RTS or DTR) to control TE. RS-485 interfaces on plug-in expansion cards generally have their own UART which also drives the TE line via the RTS or DTR (or occasionally the OUT1) lines etc. These are accessible via the normal UART registers. In some cases, custom circuits permit the receiver and transmitter to be enabled or disabled independently and these devices map the transmit-enable and receive-enable controls to different portions of the PC's I/O space: e.g. so as to overlap the UART's scratch-pad register.

Fortunately, an increasing number of RS-485 devices on the market are beginning to employ circuits which sense when the device begins to transmit and automatically enable the transceiver's transmitting element.

Because of its low cabling costs, high speed, and capability to transmit over long distances, the RS-485 standard is ideal for use in distributed control applications. It has been adopted as the basis for a number of industrial communications networks such as Profibus and Intel's Bitbus. These buses implement long-distance communication between distributed PCs and local controllers or sensors. Fieldbus

systems such as Profibus employ protocols for passing messages and data within fixed real-time constraints. So-called cyclic data transfers provide a means of implementing control loops via the network with guaranteed latency times. High transmission rates are also possible: 1.25 Mbps with Profibus. Bitbus operates in either a synchronous transmission or self-clocked mode. In the latter mode, two differential pairs are used. One carries transmitted data and the other is used for transceiver control. Self-clocking allows data to be transferred at up to 375 Kbps over distances up to approximately 300 m or 62.5 Kbps up to 1200 m. Bitbus can also operate using a synchronous protocol based on IBM's Synchronous Data Link Control (SDLC). This allows transmission at much higher rates – up to 2.4 Mbps over distances less than 300 m. The Bitbus interfaces to the PC via a dedicated adaptor unit and software drivers.

Other serial buses and standards

A number of other serial interfaces are also suitable for data acquisition, although, for PC-based applications, they are less widely used than the standards discussed previously. Standards such as the unbalanced differential RS-423 bus and RS-449 are discussed more fully in the texts by Maine (1986), Marnham (1994) and Tooley (1992).

Current loop systems

A variant of RS-232 employs current loop drivers in order to extend the maximum transmission distance. This type of interface was originally developed for driving devices such as teletypes, but several manufacturers now offer RS-232 current loop converters for use with industrial communications systems. These drivers represent the logical states within the serial character frame by the magnitude of current flowing through the loop. Most operate in the industrial standard 4–20 mA range and some allow transmission up to several kilometres (typically up to 8500 m).

The Universal Serial Bus (USB)

Intel's Universal Serial Bus is supported by a number of prominent PC and component manufacturers. It was introduced in the mid-1990s and most new PCs possess a USB controller that provides a USB root hub and two USB ports.

USB is a very high speed serial link that is capable of transfer rates rivalling some parallel buses (up to 12 Mbps). Each USB port can address up to 63 separate devices via a simple and inexpensive

4-conductor cable. Cable length is limited to 5 m for 12 Mbps transmission rates. USB devices can be daisy-chained so that in many applications the PC will require just one USB port.

Only a small number of DA&C products are currently available for the USB. Most of these are laboratory or test instruments (oscilloscopes or high precision voltmeters etc.) although this may change in the next few years as USB is implemented more widely. Because USB devices mostly require their own enclosures and external power supplies, USB implementations of simple DA&C products (digital I/O cards or simple ADC cards) may not be cost effective. In the field of data acquisition, it is likely that USB will be used first for interfacing to more complex devices. Another potential use for USB is as the primary interface between the PC and an external fieldbus or instrumentation bus such as IEEE-488.

Firewire (IEEE-1394)

Like the USB, IEEE-1394 is a relatively new development in serial buses. It is derived from a high speed supplementary serial bus intended for use in VME-based computers. IEEE-1394 (also known as Firewire) has many potential uses and implementations on PC compatible computers although, at the time of writing, most of these have yet to be realized. Microsoft's recently announced plans to use the bus in PC-based home entertainment systems may help to enhance the popularity of IEEE-1394. It is conceivable that in the long term IEEE-1394 will become the standard communications and networking interface present on the PC; possibly even replacing the RS-232 and Centronics interfaces.

The most important feature of IEEE-1394 is its capability to transfer data at very high speeds. The bus permits transmission at up to several hundred Mbps (400 Mbps and 1 Gbps in its fastest implementations) over cables up to 4.5 m long. Such rates of throughput make IEEE-1394 suitable for video disk drives and other high speed applications. Up to 63 devices can be connected on one daisy-chained network. Devices are linked via simple and relatively cheap cables which employ two double-shielded twisted-pair signal wires together with a pair of power lines.

8.5 Asynchronous serial I/O on the PC

Modern PCs are normally equipped with one or two RS-232 compatible serial ports. Some machines (particularly those of the PS/2 line) can accommodate up to four serial ports. Real-mode (e.g. DOS) programs may require drivers to be specially written because serial

I/O via the DOS file system or the BIOS is usually too slow and inflexible for data acquisition. It is less likely that Windows and OS/2 programmers will need to write serial port drivers as suitable software is available from most manufacturers of serial communications products. Whatever your interest in serial communications, it is instructive to investigate the workings of the UART as its features will have an important bearing on the capabilities of your communications software.

With the exception of one or two older PC clones, there is a great deal of standardization, in terms of UART types and addresses, between the various IBM compatible machines on the market. This greatly simplifies programming the UART and means that it is not always necessary (or desirable) to resort to the BIOS's serial port services. However, certain aspects of the serial port BIOS are useful and we will briefly discuss these before progressing to the topic of UART programming.

Serial port parameters in the BIOS Data Area

The BIOS Data Area contains a block of four words that hold the base addresses of each UART present in the system's I/O space. These are initialized by the BIOS's POST routines. On most PCs, only the first two of these ever contain valid addresses, but on some clone machines and PS/2 machines, all four may be defined. All undefined entries in this table of addresses are set to zero. The table is constructed beginning at address 0040:0000h such that the addresses of all ports are placed in contiguous positions in the table – i.e. a blank (zero) entry will never be placed between two valid UART addresses. Bits 9 to 11 of the word at 0040:0010h contain a binary-coded representation of the total number of UARTs detected by the BIOS. These addresses are summarized in Table 8.4.

In most systems the first two UARTs reside at addresses 3F8h and 2F8h in the I/O space. It is not advisable to rely on this, however, as the UARTs may be mapped to different addresses in some machines. You should always obtain the UART addresses by referring to the table at 0040:0000h as shown in Listing 8.1.

A second table in the BIOS Data Area contains the serial port timeout values that are used by the BIOS's serial port services. The table starts at 0040:007Ch and contains 1 byte for each of the four possible ports. Each byte represents a timeout interval in units of approximately 2 ms (although the actual timing will vary somewhat between different machines).

Both the address table and the timeout table will always include space for up to four entries even though, on most PC and AT

Table 8.4 *Serial port parameters in the BIOS Data Area*

Address	*Size (bytes)*	*Description*
0040:0000h	2	Serial port 1 I/O address
0040:0002h	2	Serial port 2 I/O address
0040:0004h	2	Serial port 3 I/O address
0040:0006h	2	Serial port 4 I/O address
0040:0010h	2	Bits 9–11 = Number of serial ports detected
0040:007Ch	1	Serial port 1 timeout
0040:007Dh	1	Serial port 2 timeout
0040:007Eh	1	Serial port 3 timeout
0040:007Fh	1	Serial port 4 timeout

Listing 8.1 *Determining UART addresses*

```
      :
unsigned char NumSerialPorts;
unsigned int  BaseAddr[4];
unsigned char PortNum;
      :
      :
NumSerialPorts = 0;
for (PortNum = 0; PortNum <= 3; PortNum++)
  {
  BaseAddr[PortNum] = peek(0x0040,(2 * PortNum));
  if (BaseAddr[PortNum] != 0) NumSerialPorts++;
  }
      :
```

compatible systems, the BIOS's POST routines will only search for the first two serial ports at addresses 3F8h and 2F8h.

Serial I/O using the BIOS

The BIOS services available on most modern PCs allow single characters to be transferred at up to 9600 baud via any of the available serial ports. The PS/2 BIOS permits a higher maximum (documented) transmission rate of 19 200 baud. These services do not provide interrupt driven or buffered I/O (in fact, the BIOS POST routines disable the UART's interrupts) and, because of this and the maximum transmission rate of 9600 or 19 200 baud, they are generally unsuitable for high speed I/O. However, the BIOS services (accessed via interrupt 14h) do provide a very simple means of accessing the serial ports and so this method may be preferable when throughput is not critical. The reader is referred to one of the many PC programmers'

reference books, such as Sanchez and Canton (1994), Dettmann and Johnson (1992) or van Gilluwe (1994) for more information on this topic.

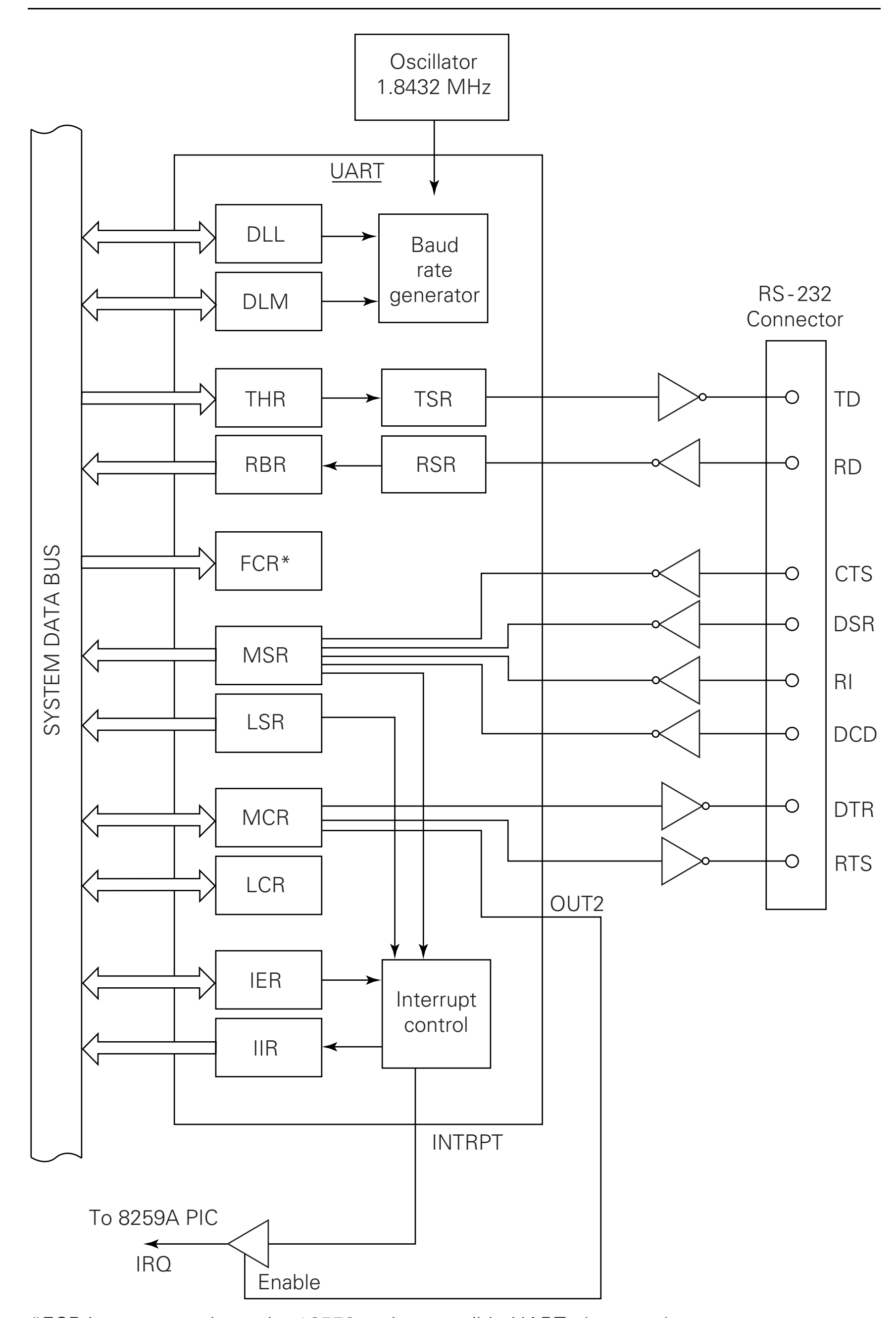
Programming the UART

The IBM PC, XT, AT, PS/2 and the various compatible machines are equipped with a range of different UARTs. The original PC used the 8-bit National Semiconductor INS 8250 UART, but most later machines possess the faster 16-bit National Semiconductor 16450 IC which in other respects is identical to the 8250 and may be programmed in the same way. Some newer models are equipped with the 16550 UART instead. This is compatible with the earlier 8250 and 16450 UARTs but also includes a facility for buffering both received data and data that is to be transmitted. The buffer holds up to 16 bytes and allows the UART to process more data before requiring service from the processor, thus reducing the software overhead. This is a particularly useful feature in multitasking and real-time systems. Note that on some machines the UART functionality is provided on the motherboard by a device such as an 82091AA integrated circuit. This is software compatible with the standard 16450 UART.

Many serial port adaptor cards that plug into the PC's expansion sockets are equipped with 16450 or 16550 UARTs. High speed industrial communications adaptors, in particular, often make use of the 16550 UART to enhance throughput. A number of other compatible UARTs are also available and these may be used in a few systems. Because most UARTs used in the PC and in PC-based DA&C systems are software compatible with the 8250, 16450 or 16550, reference will be made only to these basic UART devices in the remainder of this chapter. Serial ports based on enhanced designs (such as the 82510, 161450, 161550 and compatible devices) can also be programmed on the basis of the information supplied.

Overview of the UART and serial port

The main functional components of the PC's serial port are shown in Figure 8.8. This illustrates an RS-232 port in which an 8250 or 16450 UART is interfaced to the serial port connector via an array of inverting line drivers. In the case of an RS-422 or RS-485 port, some or all of the handshaking lines may not be connected even though they are present on the UART. As mentioned previously, an RS-485 port would employ a transceiver (see Figure 8.7), which may be enabled to transmit or receive by means of one of the unused control lines (e.g. RTS) or by additional circuitry.



*FCR is present only on the 16550 and compatible UARTs (see text).

Figure 8.8 *Main functional components of the PC's serial port*

The blocks shown along the left-hand side of the UART each represent one 8-bit register that can be read or written by software. Most of the registers are mapped permanently to different I/O addresses, but the THR and RBR both occupy the same address. Writing to this address loads a byte into the THR, while a read operation accesses the contents of the RBR. On the 16550 UART, the THR and RBR are each supplemented by a 16-byte FIFO buffer (not shown). The operation of these buffers is controlled by means of the FIFO Control Register (FCR) as described in the *FIFO Buffer Control Register* section later in this chapter. The Transmitter Shift Register (TSR) is used internally by the UART for composing the serial bit stream. The Receiver Shift Register (RSR) performs the converse function. Neither the TSR nor the RSR can be accessed directly by software.

The baud rate divisor latch registers, DLL and DLM, are mapped to the same locations as the RBR/THR and IER, respectively. DLL and DLM are accessible only when the DLAB bit in the LCR is set to 1. DLAB should be set when accessing these registers: it should always be reset to zero for normal transmission and reception of data. Table 8.5 summarizes the various registers and lists their addresses (i.e. their offsets) relative to the UART's base address.

The UART's registers

The following sections describe each of the UART's registers and the conditions under which they can be accessed. It is necessary to

Table 8.5 *8250, 16450 and 16550 UART registers*

Offset	*Mnemonic*	*Name*	*R/W*	*Notes*
0	RBR	Receiver Buffer Register	R/O	DLAB = 0
0	THR	Transmitter Holding Register	W/O	DLAB = 0
0	DLL	Divisor latch LSB	R/W	DLAB = 1
1	DLM	Divisor latch MSB	R/W	DLAB = 1
1	IER	Interrupt Enable Register	R/W	DLAB = 0
2	IIR	Interrupt Identification Register	R/O	
2	FCR	FIFO Buffer Control Register	W/O	16550 and compatibles
3	LCR	Line Control Register	R/W	DLAB is bit 7
4	MCR	Modem Control Register	R/W	
5	LSR	Line Status Register	R/O	
6	MSR	Modem Status Register	R/W	
7	SCR	Scratchpad Register	R/W	Not present on some 8250s

set DLAB to either 0 or 1 in order to read or write certain registers. Where registers can be accessed independently of the state of DLAB, it is advisable always to set DLAB to 0 in order to ensure compatibility with later devices.

Transmitter Holding Register (THR, offset 0, W/O, DLAB = 0)

This register holds the next data byte to be transmitted. Once data has been written to the THR, the UART will automatically convert it into serial format, adding the appropriate start, parity and stop bits. It will then begin transmitting the serial frame via the Transmitter Shift Register (TSR). The low order bit is transmitted first. If fewer than 8 data bits have been specified, the unused high order bits in the THR are ignored. The driving software should not attempt to load data into the THR until the THRE flag in the Line Status Register is 1.

Receiver Buffer Register (RBR, offset 0, R/O, DLAB = 0)

As the UART receives each successive data byte via its Receiver Shift Register (RSR), it strips off the start, parity and stop bits and converts the data bits into parallel format. The resulting byte is then stored in the RBR from where it can be read by the software. If fewer than 8 data bits are included in the serial frame, the high order bits in RBR are all set to 0. The DR bit in the Line Status Register is set high whenever new data is transferred into the RBR. The driving software should check the state of the DR bit and, when it is 1, the software should read the RBR. Failing to read the RBR when new data is ready will result in it being overwritten when a new byte is received. This condition, known as an overrun error, is detected by the UART and flagged by means of the OE bit in the Line Status Register.

Divisor Latch LSB (DLL, offset 0, R/W, DLAB = 1)

This register contains the least significant byte of the 16-bit divisor used to generate the required baud rate. It can be accessed only when the DLAB bit in the Line Control Register is set.

Divisor Latch MSB (DLM, offset 1, R/W, DLAB = 1)

This register contains the most significant byte of the 16-bit divisor used to generate the required baud rate. It can be accessed only when the DLAB bit in the Line Control Register is set.

Interrupt Enable Register (IER, offset 1, R/W, DLAB = 0)

The IER contains the 4 bits listed in Table 8.6. These are used to enable or disable the UART's interrupts. The UART can generate

Table 8.6 *The Interrupt Enable Register (IER)*

Bit	*Mnemonic*	*Description*
0	DRI	1 = Enable Data Ready interrupt. Interrupt occurs whenever data is available in RBR. On the 16550 if FIFO is enabled, this bit enables the character timeout interrupt.
1	THREI	1 = Enable THR Empty interrupt. Interrupt occurs when the THR is empty and ready for the next byte.
2	RLSI	1 = Enable Receiver Line Status interrupt. Interrupt occurs when framing, overrun or parity errors are detected or when Break is detected.
3	MSI	1 = Enable Modem Status interrupt. Interrupt occurs whenever CTS, DSR, RI or DCD is asserted.
7–4	–	Unused – set to 0.

interrupts as a result of several conditions and these can be selectively enabled or disabled by writing to this register. A high bit in any of the four low order positions will enable the corresponding interrupt. Any combination of interrupts can be enabled. A detailed description of the UART's interrupt system is provided later in this chapter.

Interrupt Identification Register (IIR, offset 2, R/0, DLAB = 0/1)

Once an interrupt has been generated, it is important for the interrupt handling software to be able to check the source of the interrupt in order to respond appropriately. The bits contained in the IIR (see Table 8.7) indicate, first, whether an interrupt is pending and, second, what particular UART condition generated the interrupt.

On the 16550 (or compatible UARTs), but not the 8250 or 16450, this register also contains bits that can be used to identify the type of UART and whether the FIFO buffers are enabled.

FIFO Buffer Control Register (FCR, offset 2, W/O, DLAB = 0)

This register provides a means for the software to enable and control the transmit and receive FIFO buffers which are present on the 16550 and compatible UARTs (e.g. the 16552 and 16554). The FCR is not present on the 8250 or 16450. The bit assignments for this register are shown in Table 8.8. Please refer to the later section *Operation of the 16550 FIFO buffer* for further details on interrupt generation.

Table 8.7 *The Interrupt Identification Register (IIR)*

Bit	*Mnemonic*	*Description*
0	IP	0 = Interrupt is pending
3–1	ID Table	000b = Modem Status interrupt 001b = Transmit Holding Register Empty interrupt 010b = Data Read interrupt (received data is present in RBR) 011b = Receiver Line Status interrupt 110b = Character timeout interrupt (16550 in FIFO mode)
5,4	–	Unused – should be 0
7,6	FIFO Indicator	00b = FIFO disabled, or not 16550 10b = FIFO disabled (16550 with faulty FIFO) x1b = FIFO enabled

Table 8.8 *The FIFO Buffer Control Register (FCR)*

Bit	*Mnemonic*	*Description*
0	FE	FIFO buffer enable. 0 disables and flushes both the transmit and receive FIFOs. 1 enables both FIFO buffers. NB. When writing to this register, bit 0 must be 1 in order to change the states of any of the remaining bits.
1	RRF	Reset receiver FIFO. A 1 bit empties the receiver FIFO. It has no effect on the RSR.
2	RTF	Reset transmitter FIFO. A 1 bit flushes the transmitter FIFO buffer. It has no effect on the TSR.
3	–	Unused on the PC.
4	–	Unused.
5	–	Unused.
7,6	RTL	Receiver trigger level. Specifies the number of bytes which must be available in the receiver FIFO before a Data Ready interrupt will be generated: 00b = Interrupt triggered by 1 byte in FIFO 01b = Interrupt triggered by 4 bytes in FIFO 10b = Interrupt triggered by 8 bytes in FIFO 11b = Interrupt triggered by 14 bytes in FIFO.

Line Control Register (LCR, offset 3, R/W, DLAB = 0/1)

The LCR is used to specify the composition of each serial frame. Its contents define the parity as well as the number of data and stop bits to be used, as shown in Table 8.9.

Table 8.9 *The Line Control Register (LCR)*

Bit	*Mnemonic*	*Description*
1,0	WLSB	Word length select bits. Specifies the number of data bits in the serial frame: 00b = 5 data bits 01b = 6 data bits 10b = 7 data bits 11b = 8 data bits
2	STB	Number of stop bits. This is interpreted differently, depending upon the number of data bits specified: STB = 0 always indicates 1 stop bit STB = 1 indicates 2 stop bits with 6, 7 or 8 data bits, but only 1.5 stop bits if 5 data bits have been selected.
3	PEN	Parity enable. A 1 bit enables parity. A 0 bit disables parity, regardless of the states of bit 4.
4	EPS	Even parity select. A 1 bit selects even parity. A 0 bit selects odd parity.
5	SP	Stick parity. A 1 bit forces the parity bit in the serial frame to a fixed state, regardless of whether there are an even or odd number of data bits. In this case the actual state of the parity bit is equal to the inverse of EPS.
6	SB	Set break. When this bit is set to 1, the UART forces the TD line to a spacing state. If this state is maintained for more than one character transmission period, the receiving UART will detect the break condition and (if programmed to do so) will generate a break interrupt.
7	DLAB	Divisor latch access bit. This bit should normally be 0 and only set to 1 while accessing the divisor latch (DLL and DLM) registers. It should always be reset to 0 after programming the divisor.

The Divisor Latch Access Bit (DLAB) is also contained in this register. This bit should be 0 for normal operation. It should be set to 1 only to access the baud rate divisor latches.

Modem Control Register (MCR, offset 4, R/W, DLAB = 0/1)

The primary purpose of this register is to allow driving software to control the state of the serial port's DTR and RTS lines. This is accomplished by setting or resetting bits 0 and 1 as shown in Table 8.10.

Table 8.10 *The Modem Control Register (MCR)*

Bit	*Mnemonic*	*Description*
0	DTR	1 = Assert the RS-232 DTR line.
1	RTS	1 = Assert the RS-232 RTS line.
2	OUT1	Unused on the PC.
3	OUT2	1 = Enable UART interrupts to be passed to the 8259A PIC.
4	LOOP	1 = Loop-back mode active.
7–5	–	Unused.

The OUT2 bit controls whether any interrupt signals generated by the UART reach the PC's 8259A PIC and thus cause an interrupt. OUT2 should be set high to enable UART interrupts.

The UART's loop-back facility can be enabled by setting the LOOP bit to 1. When loop-back mode is active, the modem control bits and OUT1 and OUT2 defined in the MCR are automatically fed back to the modem status input bits of the MSR. This feature is provided in order to facilitate testing.

Line Status Register (LSR, offset 5, R/O, DLAB = 0/1)

The Line Status Register contains a number of bits which indicate the status of the receiver and transmitter. These are listed in Table 8.11. The various error flags (PE, FE etc.) should be read at the time that the high DR bit is detected. These flags indicate whether any errors occurred during reception of the character currently waiting in the RBR. If the 16550's FIFO buffers are enabled, the receiver's error status is stored along with each received character in the FIFO buffer. As each new character is presented at the RBR, the UART loads the corresponding error status bits into the LSR.

Modem Status Register (MSR, offset 6, R/O, DLAB = 0/1)

The various RS-232 control lines can be sensed via the high order 4 bits of this register. In addition, the low order bits indicate whether the control lines have changed state since the last time that the software read the MSR. These are listed in Table 8.12.

Scratchpad Register (SCR, offset 7, R/W, DLAB = 0/1)

This register may be used for temporary storage of data: it is not actually used by the UART's internal circuitry and therefore the contents of this register have no effect on the functioning of the UART. It is present on most 16450-compatible UARTs.

Table 8.11 *The Line Status Register (LSR)*

Bit	*Mnemonic*	*Description*
0	DR	Data ready. A 1 bit indicates that data is available in the RBR. On a 16550 with FIFO mode enabled, a 1 bit indicates that the FIFO holds one or more bytes of data.
1	OE	1 = Overrun error occurred. Cleared by reading LSR.
2	PE	1 = Parity error occurred. Cleared by reading LSR.
3	FE	1 = Framing error occurred. Cleared by reading LSR.
4	BI	1 = Break detected. Cleared by reading LSR.
5	THRE	1 = THR is empty and ready for a new byte to be loaded. In 16550 FIFO mode, a 1 bit indicates that the transmit FIFO is empty.
6	TEMT	1 = THR and TSR are both empty. In 16550 FIFO mode, a 1 bit indicates that the TSR and transmit FIFO are both empty.
7	ERF	Error in receiver FIFO. Present only on 16550 and compatibles when FIFO mode is enabled. A 1 bit indicates that the receiver FIFO contains one or more characters for which an error occurred (i.e. framing, parity, overrun or break). Note that the error status of each received character is recorded in the FIFO and presented at the appropriate bits in the LSR each time a new character from the FIFO is presented at the RBR. If FIFO mode is unsupported or disabled, this bit is unused and is set to 0.

Table 8.12 *The Modem Status Register (MSR)*

Bit	*Mnemonic*	*Description*
0	DCTS	1 = CTS input has changed state. Cleared by reading MSR.
1	DDSR	1 = DSR input has changed state. Cleared by reading MSR.
2	TERI	1 = RI input has changed state. Cleared by reading MSR.
3	DDCD	1 = DCD input has changed state. Cleared by reading MSR.
4	CTS	1 = CTS input is asserted.
5	DSR	1 = DSR input is asserted.
6	RI	1 = RI input is asserted.
7	DCD	1 = DCD input is asserted.

Baud rate selection

The PC's UART can be configured to operate at baud rates between 2 and 115 200 baud. Its baud rate generator circuit operates by dividing down the frequency of a periodic signal provided by an

external clock. The divisor (and hence baud rate) can be modified by loading an appropriate 16-bit value into the UART's divisor latch registers DLL and DLM. The divisor, D, can be calculated from the following formula:

$$D = \frac{f_{ck}}{16 \times b} \tag{8.1}$$

where f_{ck} is the frequency of the clock and b is the desired baud rate. On the IBM PC and all compatibles (except for the PCjr), f_{ck} is 1.8432 MHz. The frequency used on the PCjr is 1.7895 MHz. Thus on all machines except for the PCjr, this equation reduces to

$$D = \frac{115\,200}{b} \tag{8.2}$$

where the maximum value of b is 115 200 (D cannot be less than 1). Table 8.13 lists the divisors necessary to generate a range of common baud rates using Equation 8.2. Note that some baud rates cannot be set exactly and that there is consequently a slight error in the timing of the serial frame when using these settings. Fortunately, the UART is generally capable of tolerating an error of up to about 5 per cent in the baud rate.

Table 8.13 *Divisors for common baud rates on the IBM PC, XT, AT, PS/2 and compatible machines*

Nominal baud rate	*Divisor*	*Error*	*Notes*
2	E100h		No practical use other than for testing and debugging.
50	900h		
75	600h		
110	417h	0.026%	
150	300h		
300	180h		
600	C0h		
1 200	60h		
2 400	30h		
4 800	18h		
9 600	0Ch		
19 200	6h		
38 400	3h		
56 000	2h	2.86%	
115 200	1h		Not available on 8250.

Listing 8.2 *Loading the divisor into the UART's Divisor Latch Registers*

```
union dbyte                  /* For accessing high and low order bytes of */
  {                                             /* the baud rate divisor */
  unsigned int  I;
  unsigned char Ch[2];
  };
union dbyte Divisor;
     :
     :
AddrDLL = BaseAddr;           /* Usually 3F8h for COM1 or 2F8h for COM2 */
AddrDLM = BaseAddr + 1;       /* Usually 3F9h for COM1 or 2F9h for COM2 */
AddrLCR = BaseAddr + 5;       /* Usually 3FDh for COM1 or 2FDh for COM2 */
     :
OrigLCR = inportb(AddrLCR);                 /* Get current state of LCR */
outportb(AddrLCR,(OrigLCR | 0x80));   /* DLAB = 1 to access baud div regs. */
outportb(AddrDLL,Divisor.Ch[0]);       /* Output LSB of baud rate divisor */
outportb(AddrDLM,Divisor.Ch[1]);       /* Output MSB of baud rate divisor */
outportb(AddrLCR,(OrigLCR & 0x7F));                          /* DLAB = 0 */
     :
     :
```

Once the required divisor has been determined, it is necessary to load it into the UART's DLL and DLM registers as shown in Listing 8.2. Note that, in this listing, the `Divisor` is defined as a `dbyte` union in order to access its high and low order bytes individually. For brevity, other variable declarations are not shown.

After defining the addresses of the various registers in the PC's I/O space, the next task is to set the DLAB bit in the Line Control Register to 1 in order to permit the divisor latch registers DLL and DLM to be accessed. Each register holds only 8 bits of the 16-bit divisor: the least significant byte is loaded into the DLL and the most significant byte into DLM. Finally, the DLAB bit in the LCR should be restored to zero.

Serial transmission errors

The UART is capable of detecting a number of different error conditions during transmission and reception of the serial bit stream. Parity errors have already been mentioned. If a parity error is detected, the UART sets the PE bit in the LSR. Two other error conditions – overrun and framing errors – are flagged in a similar way. The OE and FE bits are used for this purpose.

Overrun errors occur during reception of data if the software does not read the received data bytes from the RBR at a high enough rate. On UARTs without a FIFO, the RBR can hold only one byte of received data. The software must ensure that it reads this byte before it is overwritten by any subsequent bytes. If the byte is not read quickly enough, the UART sets the OE bit in the LSR to indicate that one or more bytes have been overwritten.

Framing errors occur if the UART cannot detect a valid stop bit. Each stop bit should consist of a logic-high pulse, but if the received data line is in a low state when the UART expects to sample a stop bit, a framing error will be generated and flagged by means of the FE bit in the LSR. Framing errors can be caused by noise on the transmission line. They might also arise if the transmitter and receiver have been erroneously programmed to operate at different baud rates.

Note that for the software to detect a parity, overrun or framing error it must read the PE, OE and FE flags from the LSR *before* it reads the received data from the RBR, since these flags are reset by the act of reading the RBR. Ideally, the routine that checks the DR flag in order to determine whether any new data is available should also record the state of PE, OE and FE at the same time.

Although these error detection facilities are very useful, they cannot detect certain types of error in the received data. If an even number of data bits in a single character frame are corrupted, due to excessive noise on the transmission line, for example, the UART will not be able to detect a parity error. A number of more robust schemes may be used to verify the integrity of received data. One such scheme is to transmit checksums or cyclic redundancy checks with each block of data sent.

One (almost) fail-safe error checking technique, which has already been mentioned, is for the receiving device to immediately retransmit each byte of received data. This can be implemented in point-to-point or looped networks and allows the transmitter to check that the echoed byte exactly matches the one originally transmitted.

Polled transmission and reception of data

The simplest, and often the fastest, method of transferring data via the UART is to continuously poll the UART's status flags. This allows the software to determine when the UART is ready to transmit a new byte, and when it has received a character over the serial link. Listing 8.3 illustrates the procedures involved.

These functions illustrate how the software should wait for the DR or THRE flags to go high before attempting to read data from the RBR or to write data to the THR, respectively. Both routines also include a facility to return to the caller after a predetermined timeout period (controlled by the global `TxTOLimit` or `RxTOLimit` variables).

Polling the RD or THRE flags provides a very fast response, particularly if the polling routines are written in assembly language. This technique is ideal if the maximum possible throughput is required and if it is feasible to dedicate the processor to polling and servicing the UART. However, the software overhead involved in

Listing 8.3 *Polled half duplex transmission and reception of data*

```
void ReadCom(unsigned char *Data, unsigned char *OE,
             unsigned char *PE, unsigned char *FE)
/* Reads the next character received by serial port. If no characters become
   available within approximately RxTOLimit milliseconds the global TxTimeout
   flag is set. If an overrun, parity or framing error is detected, ReadCom
   returns with the OE, PE or FE flags set as appropriate.
*/
{
unsigned char DataReady;
unsigned char LSR;
unsigned int  Timer;

/* Wait for DR bit to go high before reading the RBR */
Timer = 0;
do
  {
  Timer++;
  delay(1);                                           /* Delay for 1 ms */
  LSR = inportb(AddrLSR);
  DataReady = ((LSR & 0x01) == 0x01);
  }
while ((!DataReady) && (Timer < RxTOLimit));
if (DataReady)
  {
  *Data = inportb(AddrRBR);          /* Read received data byte from RBR */
  *OE   = ((LSR & 0x02) == 0x02);             /* Check for overrun error */
  *PE   = ((LSR & 0x04) == 0x04);              /* Check for parity error */
  *FE   = ((LSR & 0x08) == 0x08);             /* Check for framing error */
  }
if (Timer >= RxTOLimit) RxTimeout = 1;           /* Signal timeout error */
}



void WriteCom(unsigned char *S, unsigned char *NumCopied)
/* This writes each character contained within the ASCIIZ string S to the
   serial port's THR for transmission. If the THR does not empty within
   TxTOLimit milliseconds, this function sets the global TxTimeout flag and
   returns. The number of bytes actually copied to the THR is returned in the
   NumCopied parameter.
*/
{
unsigned int  Timer;
unsigned char THREmpty;

*NumCopied = 0;
while ((S[*NumCopied]) && !(Error.TxTimeout))
  {
  /* Check THR is empty before writing next character */
  Timer = 0;
  do
    {
    Timer++;
    delay(1);                                         /* Delay for 1 ms */
    THREmpty = ((inportb(AddrLSR) & 0x20) == 0x20);
    }
```

Listing 8.3 *(continued)*

```
  while ((!THREmpty) && (Timer < TxTOLimit));
  if (THREmpty)
         {
         outportb(AddrTHR,S[*NumCopied]);
         (*NumCopied)++;
         }
    else TxTimeout = 1;
  }
}
```

continuous polling would be impracticable in many data-acquisition applications and, in these cases, it is necessary to make use of the UART's interrupt facilities.

The UART's interrupt system

The UART is capable of generating an interrupt whenever one of a predetermined set of events occurs. This allows it to request processor service when, for example, a new character has been received.

By using interrupts in this way, rather than polling the UART's line status flags, it is possible for the software to continue with other tasks until the UART requires service. Interrupt latencies and the software overhead involved in responding to interrupts can, in a few instances, outweigh this advantage, and in order to achieve the fastest possible throughput it may be necessary to use tightly coded polling loops instead. However, most applications benefit from the interrupt facilities offered by the UART. It is feasible to use interrupt-driven, buffered I/O at baud rates up to 56 000 or even 115 200, depending upon the speed of the PC and the software it is running.

If it is possible for processes (e.g. interrupt handlers) with higher priorities than the serial port interrupt to retain control of the system for longer than the time interval between reception of successive bytes, data may be lost as a result of an overrun error. A similar problem occurs in multitasking environments, such as Microsoft Windows, which periodically disable interrupts while performing a task switch. In such operating systems interrupt latencies tend to be much longer and less predictable than under DOS. One solution to the problem is to use a hardware FIFO buffer such as that present in the 16550 UART. Note that Windows 3.1 assumes that a 16450 is present and must be specially configured to take advantage of the 16550.

The first (COM1) serial port interrupt is usually assigned to IRQ4 and the second (COM2) is assigned to IRQ3. There are no specific interrupts reserved for other UARTs present in the system

(those controlling an additional RS-422 port, for example) and these devices may be assigned to any available interrupt (IRQ) channel. The PS/2 range of computers permit different devices to share the same IRQ level and on these machines all serial ports in the system often share IRQ 4.

The 8250 and 16450 UARTs support four types of interrupt as listed in Table 8.14. The 16550 incorporates an additional interrupt facility which allows the software to read the contents of the receiver's FIFO buffer. Each type of interrupt can be enabled by setting one of the low order 4 bits of the Interrupt Enable Register (IER) – see Table 8.6. When an interrupt occurs, the interrupt handler routine must read the Interrupt Identification Register (IIR) to determine what caused the interrupt. Bits 1 to 3 of the IIR indicate the nature of the pending interrupt as shown in Table 8.7.

Table 8.14 *UART interrupts and reset actions*

Priority	*Type*	*Causes*	*Reset action*	*IIR Bits 1–3*
1	Receiver Line Status	Overrun, parity or framing errors, or break detected.	Read LSR	011b
2	Data Ready	Received data is available in RBR (DR = 1).	Read RBR	010b
2*	Data Ready	FIFO trigger level exceeded.	FIFO contents fall below trigger	010b
2*	Character Timeout	Receiver FIFO is not empty and the FIFO contents have remained static over the last four-frame period.	Read RBR	110b
3	Transmitter Holding Register Empty	THR is now empty (THRE = 1).	Read IIR or write THR	001b
4	Modem Status	Any of the DCTS, DDSR, TERI or DDCD bits of the MSR go high.	Read MSR	000b

*16550 and compatible devices only.

When it has determined the cause of the interrupt and taken whatever action is necessary, the interrupt handler must also reset or clear the interrupt. This is performed by reading or writing specific registers as detailed in Table 8.14. The software must, of course, also acknowledge the 8259A PIC through which the interrupt was generated.

Listing 8.4 illustrates how to enable all four of the UART's interrupts. As the Data Ready and Transmitter Holding Register Empty interrupts are enabled, an interrupt will be generated whenever the DR or THRE bits in the LSR are set. This allows an interrupt handler to either copy received data from the RBR to a memory buffer or to write the next character to the THR so that the UART can transmit it.

This listing includes several lines which are required to circumvent two quirks of the 8250's interrupt operation. First, the software waits for a time period equal to that required to transmit one serial frame. This period will, of course, vary with the baud rate being used. The delay is necessary because, when power is first applied to the 8250, the THRE flag will automatically be set high. When the THRE interrupt is first enabled, the high THRE flag will cause a THRE interrupt to be generated even if the THR is not empty (i.e. if previous data is still in the process of being transmitted). Waiting for a short time ensures that the UART has had sufficient time to empty the THR. Another problem arises the first time the software writes to the IER in order to set the THREI bit (i.e. to enable the THRE interrupt). On the 8250 UART, this may not actually result in the THRE interrupt being enabled. To circumvent this problem, it is necessary to write to the IER twice in succession.

Note that, on the PC, the interrupt signal from the UART is channelled through a gate which must be enabled by setting the OUT2 bit of the Modem Control Register. The 8259A PIC must

Listing 8.4 *Enabling serial port interrupts*

```
delay(FrameTime);                /* Ensure THR is empty before proceeding */
disable();                   /* Disable interrupts while configuring UART */

/* Initialize UART interrupts */
outportb(AddrMCR,0x08);            /* Enable UART interrupt via OUT2 bit */
outportb(AddrIER,0x0F);                    /* Enable all UART interrupts */
outportb(AddrIER,0x0F);         /* Bug fix for 8250 - requires two writes */

/* Clear any status bits which may already be pending */
LSR = inportb(AddrLSR);
RBR = inportb(AddrRBR);
IIR = inportb(AddrIIR);
MSR = inportb(AddrMSR);

enable();                                        /* Re-enable interrupts */
```

also be enabled to generate interrupts on the appropriate channel, although this is not shown in Listing 8.4. As mentioned previously, IRQ4 is used for ports 0 and 2 (COM1 and COM3) and IRQ3 is used for ports 1 and 3 (COM2 and COM4) on most PCs.

The UART may also generate interrupts in response to conditions such as parity, framing or overrun errors, or in response to a change occurring in the state of one of the Modem Status lines. Listing 8.5, presented at the end of this chapter, illustrates how the interrupt system allows the software to monitor for these UART conditions.

The character timeout interrupt can occur only on the 16550 UART (and compatible devices) and is not activated in Listing 8.4. This interrupt and an extension of the Data Ready interrupt are described in more detail in the following section.

Operation of the 16550 FIFO buffer

The 16550 UART, and compatible devices such as the 16552 and 16554, are equipped with a pair of 'First-In-First-Out' (FIFO) buffers. One holds data in readiness for transmission, the other stores received data. The transmitter's FIFO can be loaded with up to 16 bytes at once and the UART will then transmit these in sequence. Similarly, the receiver's FIFO can hold several bytes of received data before requiring service from the processor. This greatly reduces the software overhead involved in serial communications and enhances the rate of data throughput. The FIFO buffers allow the system greater latitude in the regularity with which the UART is serviced. This is particularly helpful if there is a possibility that high priority interrupts or task switches will temporarily block the serial port's interrupt. For these reasons, the 16550 UART is used on a number of RS-422 and RS-485 plug-in cards for industrial communication.

It is interesting to note that some proprietary serial-port adaptor cards incorporate longer FIFO buffers: typically around 8 KB. These are often used in conjunction with some form of on-board processing capability to increase data throughput while minimizing software overheads. These devices are particularly suited to transferring large blocks of data, but may be less beneficial when single bytes or short command strings are to be transmitted. In many PC-based data-acquisition systems, the 16-bit FIFOs present on the 16550 provide an optimum (and relatively cheap) way of performing buffered serial I/O.

Initializing the 16550's FIFO buffers

The 16550's FIFO buffers are unused by default – i.e. at power up, both the transmit and receive FIFOs are disabled and the device

functions in the same way as a normal 16450. In order to enable the FIFO mode of operation, it is necessary to set bit zero of the FIFO Control Register (FCR) to 1.

In order to achieve compatibility with earlier UARTs, the software should then check to ensure that the FIFO mode has indeed been enabled by reading bit 6 of the IIR. If the UART is an 8250 or 16450 device (or one of the early versions of the 16550 that happened to possess a faulty FIFO buffer), FIFO mode will not be supported and this bit will be zero. If the FIFO mode has been enabled successfully, bit 6 will be set to 1. Unfortunately, the 16550 'compatible' UART present in some AT clones (the UM82C550) does not set bit 6 even though it supports a fully working FIFO. If the driver software determines that bit 6 is zero, it is advisable to perform an additional check to determine whether the FIFO mode is actually available. This may be accomplished by switching the UART into loop-back mode and then transmitting 16 test bytes. The same sequence of bytes should be subsequently detected at the RBR if the FIFO buffer is supported. If the UART does not possess working FIFO buffers, an overrun error will occur.

When enabled, the FIFO buffers effectively replace the normal THR and RBR, buffering both transmitted and received data. The THR and RBR then act only as 'windows' through which to access the respective FIFO buffers. For simple polled operation, both transmission and reception via the FIFO buffers are performed transparently to the driving software.

Polled transmission via the FIFO

To transmit data via the FIFO buffer, the software may load up to 16 bytes at a time into the THR at offset 0 from the UART's base address, provided that the THRE flag (in the Line Status Register) is set. The UART will then transmit the bytes in sequence. When all of the bytes have been transmitted, the THRE flag will be set again to indicate that the transmitter's buffer is empty and ready for up to 16 further bytes.

According to van Gilluwe (1994), precautions should be taken if only 1 byte is to be loaded into the transmitter's FIFO. If the FIFO has *just* emptied and the last byte from the FIFO is still being transmitted via the Transmitter Shift Register (TSR), and then a single byte is loaded into the FIFO, the new byte will not be transmitted immediately. It will remain in the transmitter's FIFO until 1 or more further bytes are also loaded into the buffer. To prevent this problem occurring, it is advisable to wait until the TEMT flag in the Line Status Register is set before loading a *single*

byte into the FIFO buffer. If more than 1 byte is to be loaded, the software need not wait for the TEMT signal.

Polled reception via the FIFO

As successive bytes are received via the UART's Receiver Shift Register, they are stored, together with any error information (i.e. parity or framing errors, or a break interrupt), in the receiver's FIFO. When there are 1 or more bytes present, the DR bit in the Line Status Register is set to indicate that data can be read via the RBR. Only when all available bytes have been read from the FIFO buffer, will the UART reset the DR flag.

Interrupt-based transmission via the FIFO

Interrupt-based transmission is similar to that used on the 8250 and 16450. If the THRE interrupt is enabled, an interrupt will be generated when the transmit FIFO becomes empty, thereby allowing the software to load one or more further bytes into the buffer.

Interrupt-based reception via the FIFO

Interrupt-based reception via the FIFO is slightly more complex. A Data Ready interrupt will be generated only when a preprogrammed number of bytes are present in the receiver's FIFO. This number, known as a *Receiver Trigger Level*, may be set to 1, 4, 8 or 14 by means of the RTL bits in the FIFO Control Register. This allows the driving software to reduce the interrupt rate (and thus to enhance the system's throughput) by using a higher trigger level. The UART also provides a facility to periodically flush the receiver FIFO if there has been no FIFO activity for a time period equivalent to four serial frames. This is accomplished by another type of interrupt known as the Character Timeout Interrupt which is generated only if the FIFO is not empty and if no bytes have been added to, or read from, the receiver's FIFO during the four-frame timeout period. When the software detects a Character Timeout Interrupt, it should read the entire contents of the receiver's FIFO. This type of interrupt is cleared whenever the software reads a byte from the FIFO.

Error flagging in the FIFO

As mentioned previously, any errors which are detected in the received data byte are stored along with the data itself in the receiver's FIFO buffer. As each successive byte is presented at the RBR the associated error flags (PE, FE and BI) are also presented in the Line Status Register. If the FIFO receives more characters than it is able

to handle (i.e. more than 16 characters) it generates an overrun error which is flagged by means of the OE bit in the Line Status Register. An overrun error occurs only if the FIFO buffer is full and an additional received byte causes it to overflow.

Loop-back mode

The UART provides a loop-back facility, which is intended for testing the UART's transmit, receive and control circuits. It can also be a useful means of testing and debugging communications driver software, circumventing the need to connect the serial port to any external test equipment. The UART may be configured for loop-back operation by simply setting the LOOP bit in the Modem Control Register (MCR). This is illustrated in Listing 8.5 at the end of this chapter.

When the loop-back mode is enabled, the UART's serial output (SOUT) pin is held in the marking (inactive) state. The serial input (SIN) pin is disconnected from the UART's internal circuits and the output of the Transmitter Shift Register (TSR) is internally connected to the input of the Receiver Shift Register (RSR). In this way all 'transmitted' data is immediately received at the RSR. Similarly, the DTR, RTS, OUT1 and OUT2 pins are forced into their inactive state and the corresponding bits in the MCR are looped back internally and connected to the DSR, CTS, RI and DCD bits in the MSR. These loop connections are summarized in Table 8.15.

Note that the OUT2 pin goes high, so it is not possible to interrupt the processor while the UART is in loop-back mode. Although the UART will generate an interrupt signal if a preprogrammed interrupt condition occurs, the signal will be prevented from reaching the PC's 8259A PIC. In order to test interrupt handlers in loop-back mode, it is necessary to employ a polling loop which monitors the IP (Interrupt Pending) bit of the IIR and issues a software interrupt whenever a UART interrupt is detected. Remember that in such a test mode, the

Table 8.15 *Internal rerouting of signals in the UART's loop-back mode*

Output signal	*Input signal*
Transmitted Data (from TSR)	Received Data (input to RSR)
DTR (from MCR)	DSR (in MSR)
RTS (from MCR)	CTS (in MSR)
OUT1 (from MCR)	RI (in MSR)
OUT2 (from MCR)	DCD (in MSR)

interrupt handler should not issue an End of Interrupt instruction to the PIC!

The break facility

If the UART's receive input is held in the spacing state for a time greater than one serial frame, a Break condition is generated. The Break may be detected by polling the BI bit in the UART's Line Status Register (LSR) or by enabling the Receiver Line Status interrupt. In the latter case, upon determining that a Receiver Line Status interrupt is pending, the interrupt handler must check the BI bit. If this bit is set, a Break condition has been detected. In this case, the software should read the RBR, as the receiver will have placed a null character (all bits zero) into the RBR.

To generate a Break condition, the transmitting UART must hold its transmit line in the spacing state. This can be accomplished by setting the SB bit of the LCR for a short time (typically for a few character frames). The software should then reset SB after this interval has elapsed so that communications can resume. Note that the UART that initiated the Break cannot transmit any further characters (although it can still receive them) while the SB bit is set.

The Break facility originates from RS-232 mainframe/terminal communications systems and was designed to allow the receiving terminal to suspend the communication session. It is of limited use in data-acquisition applications, but it is possible to use it in proprietary systems to control transmission or, perhaps, to reset a network of data-logging modules.

An 8250/16450 UART driver for buffered serial I/O

This section draws upon the information presented previously to construct a suite of driver routines for use with 8250 and 16450 UARTs. The driver software, which is shown in Listing 8.5, is also compatible with enhanced UARTs such as the 16550 or 16552, but does not make use of the FIFO buffer facilities available on these devices. Neither hardware handshaking nor software flow control are supported, but these can easily be added if required.

To begin communication you should first use `InitializeCom()` to define the various serial communications parameters, and then call `OpenCom()` which initializes the UART and activates the interrupt system. At this point, you can undertake serial communications by means of the `ComCharAvail()`, `ReadCom()` and `WriteCom()` functions. These functions can be invoked independently of each other as and when required by the calling program. Each function is thoroughly commented and should be self-explanatory.

Listing 8.5 *An 8250/16450 UART driver*

```
/*
    HALF DUPLEX DRIVER FOR AT MACHINES EQUIPPED WITH 8250 AND 16450 UARTS
    ---------------------------------------------------------------------

Instructions for Use:
--------------------
1. Call InitializeCom() to define the various serial communications parameters.
2. Call OpenCom() to configure the port and begin the communications session.
3. When needed call ComCharAvail() and ReadCom() to read received characters
   via the selected serial port, or call WriteCom() to transmit characters via
   the port.
4. To terminate communications call CloseCom().

See text for more detailed instructions.

*/

#include <dos.h>
#include <stdlib.h>


/* ============================== DEFINES ================================== */

#define MaxComPort 3                        /* Supports up to 4 serial ports */
#define RxBufLim   1023                  /* Receive buffer size = 1024 bytes */
#define TxBufLim   255                   /* Transmit buffer size = 256 bytes */

#define True       1                                 /* Boolean flag values */
#define False      0                                 /*    "     "     "    */



/* ========================= DATA DECLARATIONS ============================= */

union dbyte                     /* For accessing high and low order bytes of */
  {                                                  /* the baud rate divisor */
  unsigned int  I;
  unsigned char Ch[2];
  };

struct AddrRec                                    /* UART register addresses */
  {
  unsigned int THR;                           /* Transmitter holding register */
  unsigned int RBR;                               /* Receiver buffer register */
  unsigned int DLL;             /* Divisor latch LSB register (if DLAB = 1) */
  unsigned int DLM;             /* Divisor latch MSB register (if DLAB = 1) */
  unsigned int IER;                             /* Interrupt enable register */
  unsigned int IIR;                     /* Interrupt identification register */
  unsigned int LCR;                                 /* Line control register */
  unsigned int MCR;                                /* Modem control register */
  unsigned int LSR;                                  /* Line status register */
  unsigned int MSR;                                 /* Modem status register */
  };
```

Listing 8.5 *(continued)*

```
struct SerialFrameRec                      /* Serial communications parameters */
  {
  unsigned char BaudCode;            /* 0 = 2; 1 = 50; 2 = 75 .. 14 = 115200 */
  unsigned char DataBits;  /* 0 = 5 bits; 1 = 6 bits; 2 = 7 bits; 3 = 8 bits */
  unsigned char StopBits;                           /* 0 = 1 bit; 1 = 2 bits */
  unsigned char ParityCode;   /* 0 = None; 1 = Odd; 3 = Even; 5 = Sp; 7 = Mk */
  };

struct ComRec                                      /* Serial port and PIC data */
  {
  unsigned char         PortNum;      /* Serial port number: 0 to MaxComPort */
  unsigned char         Available;  /* Set >0 if active COM port is present */
  unsigned char         IRQNum;                 /* IRQ number used, or 0xFF */
  unsigned char         IntNum;               /* Interrupt vector type code */
  unsigned int          PICAddr;       /* Base address of primary 8259A PIC */
  unsigned char         PICMask;           /* Interrupt enable mask for PIC */
  unsigned char         OrigPICMask;     /* Original int enable mask for PIC */
  unsigned char         OrigIER;                /* Original contents of IER */
  struct AddrRec        Addr;                    /* UART register addresses */
  struct SerialFrameRec SerialFrame;   /* Baud, parity, data, stop bits etc. */
  unsigned int          RxTOLimit;                 /* Receive timeout in ms */
  unsigned int          TxTOLimit;                /* Transmit timeout in ms */
  };

struct RxRec                                           /* Received data buffer */
  {
  unsigned char Buf[RxBufLim+1];                             /* Receive buffer */
  unsigned int  BufIn;              /* Index of next free location in Buf[] */
  unsigned int  BufOut;                    /* Index of oldest byte in Buf[] */
  unsigned int  Count;                          /* Number of bytes in Buf[] */
  };

struct TxRec                                        /* Transmitted data buffer */
  {
  unsigned char Buf[TxBufLim+1];                            /* Transmit buffer */
  unsigned int  BufIn;              /* Index of next free location in Buf[] */
  unsigned int  BufOut;                    /* Index of oldest byte in Buf[] */
  unsigned int  Count;                          /* Number of bytes in Buf[] */
  unsigned char Restart;                      /* Transmission restart flag */
  };

struct ErrorRec                                                 /* Error flags */
  {
  unsigned char RxOverflow;               /* Set >0 if Rx buffer overflowed  */
  unsigned char RxTimeout;                /* Set >0 if Rx data not available */
  unsigned char TxTimeout;                /* Set >0 if Tx buffer is full     */
  unsigned char BreakInt;                 /* Set >0 when Break is received   */
  unsigned char Framing;                  /* Set >0 if framing error occurs  */
  unsigned char Parity;                   /* Set >0 if parity error occurs   */
  unsigned char Overrun;                  /* Set >0 if overrun error occurs  */
  };

struct ComRec     Com;                                        /* COM port data */
struct RxRec      Rx;                                  /* Received data buffer */
struct TxRec      Tx;                               /* Transmitted data buffer */
struct ErrorRec   Error;           /* Error flags to be read/reset by caller */
```

Listing 8.5 *(continued)*

```
void interrupt (*OrigComVector)();            /* Previous interrupt handler */



/* ========================= FUNCTION PROTOTYPES =========================== */

unsigned char ComCharAvail(void);
unsigned char ReadCom(void);
void WriteCom(unsigned char *S, unsigned char *NumCopied);
void SetBreak(unsigned char Active);
void SetLoopBackMode(unsigned char Active);
void InitializeCom(unsigned char PortNum);
void OpenCom(void);
void CloseCom(void);



/* ======================= FUNCTION IMPLEMENTATIONS ======================== */


void interrupt ComIntHandler()
/* UART interrupt handler. Invoked by Transmit Holding Register Empty
   interrupt, Received Data Available interrupt or Line Status (break, parity,
   framing or overrun error) interrupt.
*/
{
unsigned char IIR;
unsigned char LSR;
unsigned char Null;

IIR = inportb(Com.Addr.IIR);
switch (IIR & 0x0F)
  {
  case 2:                                        /* THR is empty - Priority 3 */
    if (Tx.Count > 0)
           {
           /* One or more bytes are yet to be transmitted */
           outportb(Com.Addr.THR,Tx.Buf[Tx.BufOut]);
           if (Tx.BufOut < TxBufLim)
                  Tx.BufOut++;
             else Tx.BufOut = 0;
           Tx.Count--;
           Tx.Restart = False;
           }
      else Tx.Restart = True;
    break;
  case 4:                            /* Received data is available - Priority 2 */
    if (Rx.Count <= RxBufLim)
           {
           Rx.Buf[Rx.BufIn] = inportb(Com.Addr.RBR);
           if (Rx.BufIn < RxBufLim)
                  Rx.BufIn++;
             else Rx.BufIn = 0;
           Rx.Count++;
           }
      else Error.RxOverflow = True;
```

Listing 8.5 *(continued)*

```
    break;
  case 6:                    /* Overrun, parity, framing or break - Priority 1 */
    LSR = inportb(Com.Addr.LSR);
    if ((LSR & 0x10) == 0x10)
           {
           /* Break received */
           Null = inportb(Com.Addr.RBR);  /* Read and discard null character */
           Error.BreakInt = True;
           }
      else {
           if ((LSR & 0x08) == 0x08) Error.Framing = True;  /* Framing error */
           if ((LSR & 0x04) == 0x04) Error.Parity  = True;  /* Parity error  */
           if ((LSR & 0x02) == 0x02) Error.Overrun = True;  /* Overrun error */
           }
    break;
  }

/* Acknowledge interrupt by issuing a non-specific EOI to PIC(s) */
if (Com.IRQNum > 7) outportb(0xA0,0x20);
outportb(0x20,0x20);
}



unsigned char ComCharAvail()
/* Returns True if a received character is available in the Rx.Buf buffer */
{
unsigned char Avail;

disable();
Avail = (Rx.BufOut != Rx.BufIn);
enable();
return Avail;
}



unsigned char ReadCom()
/* Reads the next character from the Rx.Buf buffer. If no character becomes
   available within approx. Com.RxTOLimit milliseconds, this function sets
   the Error.RxTimeout flag and returns a Null character.
*/
{
unsigned int  Timer;
unsigned int  Cnt;
unsigned char Data;

Timer = 0;
disable();
do
  {
  disable();
  delay(1);
  enable();
  Timer++;
  Cnt = Rx.Count;
  }
```

Listing 8.5 *(continued)*

```
while ((Cnt == 0) && (Timer < Com.RxTOLimit));
if (Cnt > 0)
       {
       Data = Rx.Buf[Rx.BufOut];
       if (Rx.BufOut < RxBufLim)
              Rx.BufOut++;
         else Rx.BufOut = 0;
       Rx.Count--;
       Error.RxTimeout = False;
       }
  else {
       Data      = 0;
       Rx.BufOut = 0;
       Rx.BufIn  = 0;
       Rx.Count  = 0;
       Error.RxTimeout = True;
       }
enable();
return Data;
}



void WriteCom(unsigned char *S, unsigned char *NumCopied)
/* This function copies the ASCIIZ string S (which must contain no more than
   256 characters) into the transmission buffer, Tx.Buf, from where the UART's
   interrupt system will transmit them. If the buffer remains full for longer
   than approx. Com.TxTOLimit milliseconds, WriteCom will return with
   the Error.TxTimeout flag set. If the transmission sequence has stopped,
   this function will attempt to restart it by writing to the THR directly.
   The number of bytes successfully copied to the Tx.Buf is returned in the
   *NumCopied parameter. If no timeout has occurred, *NumCopied should be
   equal to the length of the string S.
*/
{
unsigned char I;
unsigned int  Timer;
unsigned char THREmpty;

I          = 0;
*NumCopied = 0;

disable();
while ((S[I]) && !(Error.TxTimeout))
  {
  if (Tx.Count >= Com.TxTOLimit + 1)
    {
    /* Tx.Buf is full so wait for a byte to become free, or timeout */
    Timer = 0;
    do
      {
      enable();
      Timer++;
      delay(1);
      disable();
      }
```

Listing 8.5 *(continued)*

```
    while ((Tx.Count > TxBufLim) && (Timer < Com.TxTOLimit));
    if (Timer >= Com.TxTOLimit) Error.TxTimeout = True;
    }
  if (!(Error.TxTimeout))
    {
    /* Copy the next character to Tx.Buf */
    Tx.Buf[Tx.BufIn] = S[I];
    if (Tx.BufIn < TxBufLim)
           Tx.BufIn++;
      else Tx.BufIn = 0;
    Tx.Count++;
    (*NumCopied)++;
    }
  I++;                                 /* Address next character in string S */
  }

/* If the previous transmission sequence has ended, the last THRE interrupt
   did not cause a character to be loaded into the THR and there will,
   consequently, be no more THRE interrupts to continue transmitting the new
   characters. In this case, "manually" load the first of the new characters
   into the THR to restart transmission.
*/
if ((Tx.Restart) && (Tx.Count > 0))
  {
  /* Check THR is empty before writing next character */
  Timer = 0;
  do
    {
    enable();
    Timer ++;
    delay(1);
    disable();
    THREmpty = ((inportb(Com.Addr.LSR) & 0x20) == 0x20);
    }
  while ((Timer < Com.TxTOLimit) && !(THREmpty));
  if (Tx.Count > 0)        /* Has Tx.Buf emptied while we have been waiting? */
    {                                     /* No, so restart transmission */
    if (THREmpty)
           {
           outportb(Com.Addr.THR,Tx.Buf[Tx.BufOut]);     /* Transmit new char */
           if (Tx.BufOut < TxBufLim)
                  Tx.BufOut++;
             else Tx.BufOut = 0;
           Tx.Count--;
           Tx.Restart = False;
           }
      else Tx.Restart = True;               /* Postpone transmission restart */
    }
  }
enable();
}



void SetBreak(unsigned char Active)
/* If Active = True, this function forces the TD line to a spacing state.
   If this state is maintained for more than one serial frame time, it
```

Listing 8.5 *(continued)*

```
   generates a break condition (and possibly an interrupt) in the receiver.
*/
{
unsigned char LCR;

LCR = inportb(Com.Addr.LCR);
if (Active)
       outportb(Com.Addr.LCR,(LCR | 0x40));
  else outportb(Com.Addr.LCR,(LCR & 0xBF));
}




void SetLoopBackMode(unsigned char Active)
/* This allows the UART's loopback facility to be activated. When active,
   TD is connected to RD internally and the UART's output pins are connected
   to its inputs as follows: DTR-->DSR; RTS-->CTS; OUT1-->RI; OUT2-->DCD.
   This mode is used only for debugging and UART testing.
*/
{
unsigned char MCR;

MCR = inportb(Com.Addr.MCR);
if (Active)
       outportb(Com.Addr.MCR,(MCR | 0x10));
  else outportb(Com.Addr.MCR,(MCR & 0xEF));
}




void InitializeCom(unsigned char PortNum)
/* This initializes the Com structure for the specified serial port.
   PortNum = 0 refers to COM1, PortNum = 1 refers to COM2 etc. PortNum should
   not exceed MaxComPort. Default IRQ and register addresses are defined
   automatically. If you are using a non-standard IRQ level you will need to
   redefine the appropriate variables manually. InitializeCom also defines
   default communications parameters: 9600 baud, 8 data, 1 stop, even parity.
   Again, these can be modified, if required, before calling OpenCom().
*/
{
unsigned int BaseAddr;

if (PortNum <= MaxComPort)
       {
       BaseAddr = peek(0x40,(2 * PortNum));
       if (BaseAddr != 0)                                /* Does port exist? */
              {
              Com.PortNum   = PortNum;
              Com.Available = True;
              switch(PortNum)
                {
                case 0:  Com.IRQNum = 4; break;                      /* COM1 */
                case 1:  Com.IRQNum = 3; break;                      /* COM2 */
                case 2:  Com.IRQNum = 4; break;                      /* COM3 */
                case 3:  Com.IRQNum = 3; break;                      /* COM4 */
                }
```

Listing 8.5 *(continued)*

```
              Com.IntNum  = 8 + Com.IRQNum;
              Com.PICAddr = 0x20;
              Com.PICMask = 0x01 << Com.IRQNum;
              Com.Addr.THR = BaseAddr;
              Com.Addr.RBR = BaseAddr;
              Com.Addr.DLL = BaseAddr;
              Com.Addr.DLM = BaseAddr + 1;
              Com.Addr.IER = BaseAddr + 1;
              Com.Addr.IIR = BaseAddr + 2;
              Com.Addr.LCR = BaseAddr + 3;
              Com.Addr.MCR = BaseAddr + 4;
              Com.Addr.LSR = BaseAddr + 5;
              Com.Addr.MSR = BaseAddr + 6;
              Com.SerialFrame.BaudCode   = 10;                  /* 9600 baud */
              Com.SerialFrame.DataBits   = 3;                 /* 8 data bits */
              Com.SerialFrame.StopBits   = 0;                  /* 1 stop bit */
              Com.SerialFrame.ParityCode = 3;                 /* Even parity */
              Com.RxTOLimit = 2000;                   /* Approx. 2.0 seconds */
              Com.TxTOLimit = 100;                         /* Approx. 100 ms */
              }
         else Com.Available = False;
       }
  else Com.Available = False;
}



void OpenCom()
/* OpenCom() prepares the system for serial communication. This function must
   be called before any communication can take place. It initializes the
   Rx and Tx buffers, the UART and the PIC according to the values previously
   stored in the Com structure. For this reason all fields within Com must be
   properly initialized (by calling InitializeCom()) before OpenCom() is
   invoked.
*/
{
unsigned char MSR;
unsigned char LSR;
unsigned char RBR;
unsigned char IIR;
union dbyte   Divisor;
unsigned char Settings;

if ((Com.Available) && (Com.IRQNum < 16))
  {
  /* Initialize the Rx and Tx buffers */
  Rx.BufIn   = 0;
  Rx.BufOut  = 0;
  Rx.Count   = 0;
  Tx.BufIn   = 0;
  Tx.BufOut  = 0;
  Tx.Count   = 0;
  Tx.Restart = True;

  /* Initialize Error status record */
  Error.RxOverflow = False;
  Error.RxTimeout  = False;
```

Listing 8.5 *(continued)*

```
Error.TxTimeout  = False;
Error.BreakInt   = False;
Error.Parity     = False;
Error.Overrun    = False;
Error.Framing    = False;

/* Setup baud rate, parity, data bits and stop bits */
switch (Com.SerialFrame.BaudCode)
  {
  case  0: Divisor.I = 0xE100; break;    /* Debugging */    /* 2      baud */
  case  1: Divisor.I = 0x0900; break;                       /* 50     baud */
  case  2: Divisor.I = 0x0600; break;                       /* 75     baud */
  case  3: Divisor.I = 0x0417; break;                       /* 110    baud */
  case  4: Divisor.I = 0x0300; break;                       /* 150    baud */
  case  5: Divisor.I = 0x0180; break;                       /* 300    baud */
  case  6: Divisor.I = 0x00C0; break;                       /* 600    baud */
  case  7: Divisor.I = 0x0060; break;                       /* 1200   baud */
  case  8: Divisor.I = 0x0030; break;                       /* 2400   baud */
  case  9: Divisor.I = 0x0018; break;                       /* 4800   baud */
  case 10: Divisor.I = 0x000C; break;                       /* 9600   baud */
  case 11: Divisor.I = 0x0006; break;                       /* 19200  baud */
  case 12: Divisor.I = 0x0003; break;                       /* 38400  baud */
  case 13: Divisor.I = 0x0002; break;                       /* 56000  baud */
  case 14: Divisor.I = 0x0001; break;                       /* 115200 baud */
  default: Divisor.I = 0x000C; break;                       /* 9600   baud */
  }
Settings = ((Com.SerialFrame.ParityCode << 3) & 0x38) |
           ((Com.SerialFrame.StopBits << 2) & 0x04) |
           (Com.SerialFrame.DataBits & 0x03);
outportb(Com.Addr.LCR,0x80);           /* DLAB=1 to access baud div. regs. */
outportb(Com.Addr.DLL,Divisor.Ch[0]);            /* Output LSB of divisor */
outportb(Com.Addr.DLM,Divisor.Ch[1]);            /* Output MSB of divisor */
outportb(Com.Addr.LCR,Settings);          /* Output settings & reset DLAB */

disable();                                 /* Disable hardware interrupts */

/* Initialize UART interrupts */
/* The value loaded into the IER determines */
/* interrupts are enabled */
outportb(Com.Addr.MCR,0x08);              /* Enable UART int via OUT2 bit */
Com.OrigIER = inportb(Com.Addr.IER);
outportb(Com.Addr.IER,0x0F);                /* Enable all UART interrupts */
outportb(Com.Addr.IER,0x0F);         /* Bug fix for 8250 - needs 2 writes */

/* Clear any status bits pending by reading registers */
LSR = inportb(Com.Addr.LSR);
RBR = inportb(Com.Addr.RBR);
IIR = inportb(Com.Addr.IIR);      /* <-- This line is also an 8250 bug fix */
MSR = inportb(Com.Addr.MSR);             /* in case loading IER previously */
                                            /* generated a false THRE int. */
/* Install int handler */
OrigComVector = getvect(Com.IntNum);               /* Save original vector */
setvect(Com.IntNum,ComIntHandler);                      /* Redirect vector */

/* Update PIC's interrupt enable mask */
Com.OrigPICMask = inportb(Com.PICAddr+1);          /* Get original PIC mask */
outportb(Com.PICAddr+1,
```

Listing 8.5 *(continued)*

```
          (Com.OrigPICMask &  Com.PICMask));          /* Enable UART's IRQ */

  enable();                                    /* Re-enable hardware interrupts */
  }
}



void CloseCom()
/* This closes down the UART interrupt system and restores the original
   interrupt vector. CloseCom() must be called before the program terminates.
*/
{
disable();                                       /* Disable hardware interrupts */
outportb(Com.PICAddr+1,Com.OrigPICMask);                  /* Disable UART's IRQ */
setvect(Com.IntNum,OrigComVector);               /* Restore original int vector */
outportb(Com.Addr.IER,Com.OrigIER);              /* Restore UART's original IER */
enable();                                      /* Re-enable hardware interrupts */
}
```

Periodically, and after each call to `ReadCom()` and `WriteCom()`, you should examine the various fields of the `Error` structure to detect events such as buffer overflows, timeouts, break interrupts or parity, framing and overrun errors. Note that, for illustrative purposes, break conditions and overrun, parity and framing errors are recorded in a single global `Error` structure by the interrupt handler in Listing 8.5. Often, however, this is not the best way of detecting such error conditions, because the point at which the calling program detects that one of the `Error` flags has been set will not necessarily fall correctly in sequence with the character stream retrieved from the `Rx.Buf` buffer. `Rx.Buf` may hold, for example, 10 unread characters at the time that the interrupt handler detects an error in the 11th character. The resulting error flag might be retrieved by the caller before it has read the previous 10 correctly received characters. If you wish to preserve the temporal relationship between detection of the error flags and reception of each individual character, you should convert each entry in the `Rx.Buf` into a structure containing both data and error code fields. The UART's PE, OE, FE and BI flags must then be recorded along with each received character in the receive buffer, `Rx.Buf`.

To terminate a communications session your program should call `CloseCom()`. This function *must* be called at some point before the application terminates in order to restore the interrupt vector and disable the UART's interrupt system.

Note that `InitializeCom()` defines a set of default values for the serial parameters. You may need to modify the interrupt parameters (`IRQNum`, `IntNum`, `PICMask` and `PICAddr`) if you are working with

a non-standard hardware configuration. Different serial frame parameters can easily be substituted by changing the `BaudCode`, `DataBits`, `StopBits`, and `ParityCode` fields of the `SerialFrameRec` structure. If you need to modify any of these variables, you should do so after calling `InitializeCom()`, but before invoking `OpenCom()`.

Part 4 Interpreting and Using Acquired Data

This Page Intentionally Left Blank

9 Scaling and linearization

The task of a data-acquisition program is to determine values of one or more physical quantities, such as temperature, force or displacement. We have seen in Chapter 3 that this is accomplished by reading digitized representations of those values from an ADC. In order for the user, as well as the various elements of the data-acquisition system, to correctly interpret the readings, the program must convert them into appropriate 'real-world' units. This obviously requires a detailed knowledge of the characteristics of the sensors and signal-conditioning circuits used. The relationship between a physical variable to be measured (the measurand) and the corresponding transduced and digitized signal may be described by a response curve such as that shown in Figure 9.1.

Each component of the measuring system contributes to the shape and slope of the response curve. The transducer itself is, of course, the principal contributor, but the characteristics of the associated signal-conditioning and ADC circuits also have an important part to play in determining the form of the curve.

In some situations the physical variable of interest is not measured directly: it may be inferred from a related measurement instead. We might, for example, measure the *level* of liquid in a vessel in order to determine its volume. The response curve of the measurement system would, in this case, also include the factors necessary for conversion between level and volume.

Most data-acquisition systems are designed to exhibit linear responses. In these cases either all elements of the measuring system will have linear response curves, or they will have been carefully combined so as to cancel out any non-linearities present in individual components.

Some transducers are inherently non-linear. Thermocouples and resistance temperature detectors are prime examples, but many

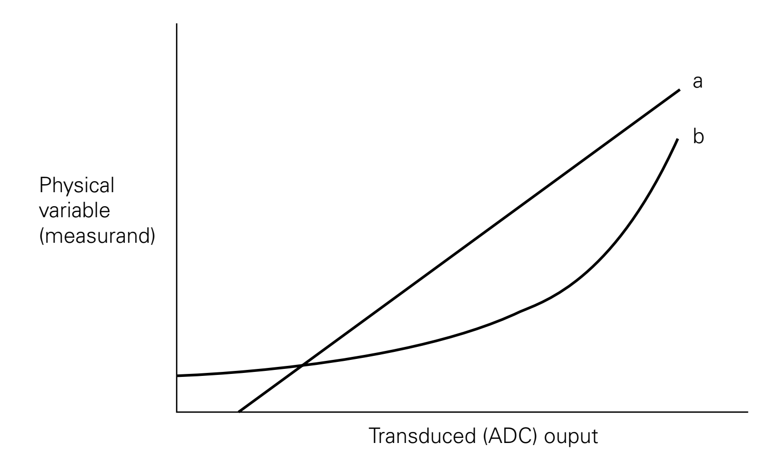


Figure 9.1 *Response curves for typical measuring systems: (a) linear response and (b) non-linear response*

other types of sensor exhibit some degree of non-linearity. Non-linearities may, occasionally, arise from the way in which the measurement is carried out. If, in the volume-measurement example mentioned above, we have a cylindrical vessel, the quantity of interest (the volume of liquid) would be directly proportional to the level. If, on the other hand, the vessel had a hemispherical shape, there would be a non-linear relationship between fluid level and volume. In these cases, the data-acquisition software will usually be required to compensate for the geometry of the vessel when converting the ADC reading to the corresponding value of the measurand.

To correctly interpret digitized ADC readings, the data-acquisition software must have access to a set of calibration parameters that describe the response curve of the measuring system. These parameters may exist either as a table of values or as a set of coefficients of an equation that expresses the relationship between the physical variable and the output from the ADC. In order to compile the required calibration parameters, the system must usually sample the ADC output for a variety of known values of the measurand. The resulting calibration reference points can then be used as the basis of one of the scaling or linearization techniques described in this chapter.

9.1 Scaling of linear response curves

The simplest and, fortunately, the most common type of response curve is a straight line. In this case the software need only be programmed with the parameters of the line for it to be able to

convert ADC readings to a meaningful physical value. In general, any linear response curve may be represented by the equation

$$y - y_0 = s(x - x_0) \tag{9.1}$$

where y represents the physical variable to be measured and x is the corresponding digitized (ADC) value. The constant y_0 is any convenient reference point (usually chosen to be the lower limit of the range of y values to be measured), x_0 is the value of x at the intersection of the line $y = y_0$ with the response curve (i.e. the ADC reading at the lower limit of the measurement range) and s represents the gradient of the response curve.

Many systems are designed to measure over a range from zero up to some predetermined maximum value. In this case, y_0 can be chosen to be zero. In all instances y_0 will be a known quantity. The task of calibrating and scaling a linear measurement system is then reduced to determining the scaling factor, s, and offset, x_0.

The offset

The offset, x_0, can arise in a variety of ways. One of the most common is due to drifts occurring in the signal-conditioning circuits as a result of variations in ambient temperature. There are many other sources of offset in a typical measuring system. For example, small errors in positioning the body of a displacement transducer in a gauging jig will shift the response curve and introduce a degree of offset. Similarly, a poorly mounted load cell might suffer transverse stresses which will also distort the response curve.

As a general rule, x_0 should normally be determined each time the measuring system is calibrated. This can be accomplished by reading the ADC while a known input is applied to the transducer. If the offset is within acceptable limits it can simply be subtracted from subsequent ADC readings as shown by Equation 9.1. Very large offsets are likely to compromise the performance of the measuring system (e.g. limit its measuring range) and might indicate faults such as an incorrectly mounted transducer or maladjusted signal-conditioning circuits. It is wise to design data-acquisition software so that it checks for this eventuality and warns the operator if an unacceptably large offset is detected.

Some signal-conditioning circuits provide facilities for manual offset adjustment. Others allow most or all of the physical offset to be cancelled under software control. In the latter type of system the offset might be adjusted (or compensated for) by means of the output from a digital-to-analogue converter (DAC). The DAC voltage

might, for example, be applied to the output from a strain-gauge-bridge device (e.g. a load cell) in order to cancel any imbalances present in the circuit.

Scaling from known sensitivities

If the characteristics of every component of the measuring system are accurately known it might be possible to calculate the values of s and x_0 from the system design parameters. In this case the task of calibrating the system is almost trivial. The data-acquisition software (or calibration program) must first establish the value of the ADC offset, x_0, as described in the preceding section, and then determine the scaling factor, s. The scaling factor can be supplied by the user via the keyboard or data file, but, in some cases, it is simpler for the software to calculate s from a set of measuring-system parameters typed in by the operator.

An example of this method is the calibration of strain-gauge-bridge transducers such as load cells. The operator might enter the design sensitivity of the load cell (in millivolts output per volt input at full scale), the excitation voltage supplied to the input of the bridge and the full-scale measurement range of the sensor. From these parameters the calibration program can determine the voltage that would be output from the bridge at full scale, and knowing the characteristics of the signal-conditioning and ADC circuits it can calculate the scaling factor.

In some instances it may not be possible for the gain (and other operating parameters) of the signal-conditioning amplifier(s) to be determined precisely. It is then necessary for the software to take an ADC reading while the transducer is made to generate a known output signal. The obvious (and usually most accurate) method of doing this is to apply a fixed input to the transducer (e.g. force in the case of a load cell). This method, referred to as prime calibration, is the subject of the following section. Another way of creating a known transducer output is to disturb the operation of the transducer itself in some way. This technique is adopted widely in devices, such as load cells, which incorporate a number of resistive strain gauges connected in a Wheatstone bridge. A shunt resistor can be connected in parallel with one arm of the bridge in order to temporarily unbalance the circuit and simulate an applied load. This allows the sensitivity of the bridge (change in output voltage divided by the change in 'gauge' resistance) to be determined, and then the ADC output at this simulated load can be measured in order to calculate the scaling factor. In this way the scaling factor will encompass the gain of the signal-conditioning circuit as well as

the conversion characteristics of the ADC and the sensitivity of the bridge itself.

This calibration technique can be useful in situations, as might arise with load measurement, where it is difficult to generate precisely known transducer inputs. However, it does not take account of factors, resulting from installation and environmental conditions, which might affect the characteristics of the measuring system. In the presence of such influences this method can lead to serious calibration errors.

To illustrate this point we will continue with the example of load cells. The strain gauges used within these devices have quite small resistances (typically less than 350 Ω). Consequently, the resistance of the leads which carry the excitation supply can result in a significant voltage drop across the bridge and a proportional lowering of the output voltage. Some signal-conditioning circuits are designed to compensate for these voltage drops, but without this facility it can be difficult to determine the magnitude of the loss. If not corrected for, the voltage drop can introduce significant errors into the calibration.

In order to account for every factor which contributes to the response of the measurement system it is usually necessary to calibrate the *whole* system against some independent reference. These methods are described in the following sections.

Two- and three-point prime calibration

Prime calibration involves measuring the input, y, to a transducer (e.g. load, displacement or temperature) using an independent calibration reference and then determining the resulting output, x, from the ADC. Two (or sometimes three) points are obtained in order to calculate the parameters of the calibration line. In this way the calibration takes account of the behaviour of the measuring system as a whole, including factors such as signal losses in long cables.

By determining the offset value, x_0, we can establish one point on the response curve – i.e. (x_0, y_0). It is necessary to obtain at least one further reference point, (x_1, y_1), in order to uniquely define the straight-line response curve. The scaling factor may then be calculated from

$$s = \frac{y_1 - y_0}{x_1 - x_0} \tag{9.2}$$

Some systems, particularly those which incorporate bipolar transducers (i.e. those which measure either side of some zero level) do not use the offset point, (x_0, y_0), for calculating s. Instead, they

obtain a reading on each side of the zero point and use these values to compute the scaling factor. In this case, y_0 might be chosen to represent the centre (zero) value of the transducer's working range and x_0 would be the corresponding ADC reading.

Accuracy of prime calibration

The values of s and x_0 determined by prime calibration are needed to convert all subsequent ADC readings into the corresponding 'real-world' value of the measurand. It is, therefore, of paramount importance that the values of s and x_0, and the (x_0, y_0) and (x_1, y_1) points used to derive them, are accurate.

Setting aside any sampling and digitization errors (see Chapters 3 and 4) there are several potential sources of inaccuracy in the (x, y) calibration points. Random variations in the ADC readings might be introduced by electrical noise or instabilities in the physical variable being measured (e.g. positioning errors in a displacement-measuring system).

Electrical noise can be particularly problematic where low level transducer signals (and high amplifier gains) are used. This is often the case with thermocouples and strain-gauge bridges, which generate only low level signals (typically several mV). Noise levels should always be minimized at source by the use of appropriate shielding and grounding techniques. Small amplitudes of residual noise may be further reduced by using suitable software filters (see Chapter 4). A simple 8× averaging filter can often reduce noise levels by a factor of 3 or more, depending, of course, upon the sampling rate and the shape of the noise spectrum.

An accurate prime calibration reference is also essential. Inaccurate reference devices can introduce both systematic and random errors. Systematic errors are those arising from a consistent measurement defect in the reference device, causing, for example, all readings to be too large. Random errors, on the other hand, result in readings that have an equal probability of being too high or too low and arise from sources such as electrical noise. Any systematic inaccuracies will tend to be propagated from the calibration reference into the system being calibrated and steps should, therefore, be taken to eliminate all sources of systematic inaccuracy. In general, the reference device should be considerably more precise (preferably at least 2 to 5 times more precise) than the required calibration accuracy. Its precision should be maintained by periodic recalibration against a suitable primary reference standard.

When calibrating any measuring system it is important to ensure that the conditions under which the calibration is performed match,

as closely as possible, the actual working conditions of the transducer. Many sensors (and signal-conditioning circuits) exhibit changes in sensitivity with ambient temperature. LVDTs, for example, have typical sensitivity temperature coefficients of about 0.01 per cent/°C or more. A temperature change of about 10°C, which is not uncommon in some applications, can produce a change in output comparable to the transducer's non-linearity. Temperature *gradients* along the body of an LVDT can have an even more pronounced effect on the sensitivity (and linearity) of the transducer.

Most transducers also exhibit some degree of non-linearity, but in many cases, if the device is used within prescribed limits, this will be small enough for the transducer to be considered linear. This is usually the case with LVDTs and load cells. Thermocouples and resistance temperature detectors (RTDs) are examples of non-linear sensors, but even these can be approximated by a linear response curve over a limited working range. Whatever the type of transducer, it is always advisable to calibrate the measuring system over the same range as will be used under normal working conditions in order to maximize the accuracy of calibration.

Multiple-point prime calibration

If only two or three (x, y) points on the response curve are obtained, any random variations in the transducer signal due to noise or positioning uncertainties can severely limit calibration accuracy. The effect of random errors can be reduced by statistically averaging readings taken at a number of different points on the response curve. This approach has the added advantage that the calibration points are more equally distributed across the whole measurement range. Transducers such as the LVDT tend to deviate from linearity more towards the end of their working range, and with two- or three-point calibration schemes this is precisely where the calibration reference points are usually obtained. The scaling factor calculated using Equation 9.1 can, in such cases, differ slightly (by up to about 0.1 per cent for LVDTs) from the average gradient of the response curve. This difference can often be reduced by a significant factor if we are able to obtain a more representative line through the response curve.

In order to fit a representative straight line to a set of calibration points we will use the technique of least-squares fitting. This technique can be used for fitting both straight lines and non-linear curves. The straight-line fit which is discussed below is a simple case of the more general polynomial least-squares fit described later in this chapter.

It is assumed in this method that there will be some degree of error in the y_i values of the calibration points and that any errors in the corresponding x_i values will be negligible, which is usually the case in a well-designed measuring system. The basis of the technique is to mathematically determine the parameters of the straight line that passes as closely as possible to each calibration point. The best-fit straight line is obtained when the sum of the squares of the deviations between all of the y_i values and the fitted line is least. A simple mathematical analysis shows that the best-fit straight line, $y = sx + h$, is described by the following well-known equations.

$$h = \frac{\sum_{i=1}^{i=n} y_i \sum_{i=1}^{i=n} x_i^2 - \sum_{i=1}^{i=n} x_i \sum_{i=1}^{i=n} x_i y_i}{\Omega}$$

$$s = \frac{n \sum_{i=1}^{i=n} x_i y_i - \sum_{i=1}^{i=n} x_i \sum_{i=1}^{i=n} y_i}{\Omega}$$

$$\delta h = \frac{\sqrt{\alpha^2 \sum_{i=1}^{i=n} x_i^2}}{\Omega} \tag{9.3}$$

$$\delta s = \frac{\alpha^2 n}{\Omega}$$

where

$$\Omega = n \sum_{i=1}^{i=n} x_i^2 - \sum_{i=1}^{i=n} x_i \sum_{i=1}^{i=n} x_i$$

$$\alpha^2 = \frac{\sum_{i=1}^{i=n} [y_i - y'(x_i)]^2}{n-2}$$

In these equations s is the scaling factor (or gradient of the response curve) and h is the transducer input required to produce an ADC reading (x) of zero. The δs and δh values are the uncertainties in s and h, respectively. It is assumed that there are n of the (x_i, y_i) calibration points.

Listing 9.1 *C function for performing a first order polynomial (linear) least-squares fit to a set of calibration reference points*

```
#include <math.h>

#define True    1
#define False   0
#define MaxNP   500                 /* Maximum number of data points for fit */

struct LinFitResults         /* Results record for PerformLinearFit function */
  {
  double Slope;
  double Intercept;
  double ErrSlope;
  double ErrIntercept;
  double RMSDev;
  double WorstDev;
  double CorrCoef;
  };

struct LinFitResults LResults;
unsigned int         NumPoints;
double               X[MaxNP];
double               Y[MaxNP];

void PerformLinearFit()
/* Performs a linear (first order polynomial) fit on the X[],Y[] data points
   and returns the results in the LResults structure.
*/
{
unsigned int I;
double   SumX;
double   SumY;
double   SumXY;
double   SumX2;
double   SumY2;
double   DeltaX;
double   DeltaY;
double   Deviation;
double   MeanSqDev;
double   SumDevnSq;

SumX  = 0;
SumY  = 0;
SumXY = 0;
SumX2 = 0;
SumY2 = 0;

for (I = 0; I < NumPoints; I++)
  {
  SumX  = SumX  + X[I];
  SumY  = SumY  + Y[I];
  SumXY = SumXY + X[I] * Y[I];
  SumX2 = SumX2 + X[I] * X[I];
  SumY2 = SumY2 + Y[I] * Y[I];
  }
DeltaX    = (NumPoints * SumX2) - (SumX * SumX);
DeltaY    = (NumPoints * SumY2) - (SumY * SumY);
```

Listing 9.1 *(continued)*

```
LResults.Intercept = ((SumY * SumX2) - (SumX * SumXY)) / DeltaX;
LResults.Slope     = ((NumPoints * SumXY) - (SumX * SumY)) / DeltaX;

SumDevnSq          = 0;
LResults.WorstDev = 0;
for (I = 0; I < NumPoints; I++)
  {
  Deviation = Y[I] - (LResults.Slope * X[I] + LResults.Intercept);
  if (fabs(Deviation) > fabs(LResults.WorstDev)) LResults.WorstDev = Deviation;
  SumDevnSq = SumDevnSq + (Deviation * Deviation);
  }
MeanSqDev = SumDevnSq / (NumPoints - 2);
LResults.ErrIntercept = sqrt(SumX2 * MeanSqDev / DeltaX);
LResults.ErrSlope     = sqrt(NumPoints * MeanSqDev / DeltaX);
LResults.RMSDev       = sqrt(MeanSqDev);
LResults.CorrCoef     = ((NumPoints * SumXY) - (SumX * SumY)) /
                                                       sqrt(DeltaX * DeltaY);
}
```

These formulae are the basis of the `PerformLinearFit()` function in Listing 9.1. The various summations are performed first and the results are then used to calculate the parameters of the best-fit straight line. The `Intercept` variable is equivalent to the quantity h in the above formulae while `Slope` is the same as the scaling factor, s. The `ErrIntercept` and `ErrSlope` variables are equivalent to δh and δs, and may be used to determine the statistical accuracy of the calibration line. The function also determines the conformance between the fitted line and the calibration points and then calculates the root-mean-square (rms) deviation (the same as α^2) and worst deviation between the line and the points.

It is always advisable to check the rms and worst deviation figures when the fitting procedure has been completed, as these provide a measure of the accuracy of the fit. The rms deviation may be thought of as the average deviation of the calibration points from the straight line.

The ratio of the worst deviation to the rms deviation can indicate how well the calibration points can be modelled by a straight line. As a rule-of-thumb, if the worst deviation exceeds the rms deviation by more than a factor of about 3 this might indicate one of two possibilities: either the true response curve exhibits a significant non-linearity or one (or more) of the calibration points has been measured inaccurately. Any uncertainties from either of these two sources will be reflected in the `ErrorIntercept` and `ErrorSlope` variables.

Although there is a potential for greater accuracy with multiple-point calibration, it should go without saying that the comments made in the preceding section, concerning prime-calibration accuracy, also apply to multiple-point calibration schemes.

To minimize the effect of random measurement errors, multiple-point calibration is generally to be preferred. However, it does have one considerable disadvantage: the additional time required to carry out each calibration. If a transducer is to be calibrated *in situ* (while attached to a machine on a production line, for example) it can sometimes require a considerable degree of effort to apply a precise reference value to the transducer's input. Some applications might employ many tens (or even hundreds) of sensors and recalibration can then take many hours to complete, resulting in project delays or lost production time. In these situations it may be beneficial to settle for the slightly less accurate two- or three-point calibration schemes. It should also be stressed that two- and three-point calibrations *do* often provide a sufficient degree of precision and that multiple-point calibrations are generally only needed where highly accurate measurements are the primary concern.

Applying linear scaling parameters to digitized data

Once the scaling factor and offset have been determined they must be applied to all subsequent digitized measurements. This usually has to be performed in real time and it is therefore important to minimize the time taken to perform the calculation. Obviously, high speed computers and numeric coprocessors can help in this regard, but there are two ways in which the efficiency of the scaling algorithm can be enhanced.

First, floating-point multiplication is generally faster than division. For example, Borland Pascal's floating-point routines will multiply two real type variables in about one-third to one-half of the time that they would take to carry out a floating-point division. A similar difference in execution speeds occurs with the corresponding 80x87 numeric coprocessor instructions. Multiplicative scaling factors should, therefore, always be used – i.e. always multiply the data by s, rather than dividing by s^{-1} – even if the software specification requires that the inverse of the scaling factor is presented on displays and printouts etc.

Second, the scaling routines can be coded in assembly language. This is simpler if a numeric coprocessor is available, otherwise floating-point routines will have to be specially written to perform the scaling.

In very high speed applications, the only practicable course of action might be to store the digitized ADC values directly into RAM and to apply the scaling factor(s) after the data-acquisition run has been completed, when timing constraints may be less stringent.

9.2 Linearization

Linearization is the term applied to the process of correcting the output of an ADC in order to compensate for non-linearities present in the response curve of a measuring system. Non-linearities can arise from a number of different components, but it is often the sensors themselves that are the primary sources.

In order to select an appropriate linearization scheme, it obviously helps to have some idea of the shape of the response curve. The response of the system might be known, as is the case with thermocouples and RTDs. It might even conform to some recognized analytical function. In some applications the deviation from linearity might be smooth and gradual, but in others, the non-linearities might consist of small-scale irregularities in the response curve. Some measuring systems may also exhibit response curves that are discontinuous or, at least, discontinuous in their first and higher order derivatives.

There are several linearization methods to choose from and whatever method is selected, it must suit the peculiarities of the system's response curve. Polynomials can be used for linearizing smooth and slowly varying functions, but are less suitable for correcting irregular deviations or sharp 'corners' in the response curve. They can be adapted to closely match a known functional form or they can be used in cases where the form of the response function is indeterminate. Interpolation using look-up tables is one of the simplest and most powerful linearization techniques and is suitable for both continuous and discontinuous response curves. Each method has its own advantages and disadvantages in particular applications and these are discussed in the following sections.

The capability to linearize response curves in software can, in some cases, mean that simpler and cheaper transducers or signal-conditioning circuitry can be used. One such case is that of LVDT displacement transducers. These devices operate rather like transformers. An AC excitation voltage is applied to a primary coil and this induces a signal in a pair of secondary windings. The degree of magnetic flux linkage and, therefore, the output from each of the secondary coils is governed by the linear displacement of a ferrite core along the axis of the windings. In this way, the output from the transducer varies in relation to the displacement of the core.

Simple LVDT designs employ parallel-sided cylindrical coils. However, these exhibit severe non-linearities (typically up to about 5 or 10 per cent) as the ferrite core approaches the ends of the coil assembly. The non-linearity can be corrected in a variety of ways, one of which is to layer windings in a series of steps towards the

ends of the coil. This can reduce the overall non-linearity to about 0.25 per cent. It does, however, introduce additional small-scale non-linearities (of the order of 0.05 to 0.10 per cent) at points in the response curve corresponding to each of the steps.

It is a relatively simple matter to compensate for the large-scale non-linearities inherent in parallel-coil LVDT geometries by using the polynomial linearization technique discussed in the following section. Thus, software linearization techniques allow cheaper LVDT designs to be used and this has the added advantage that no small-scale (stepped winding) irregularities are introduced. This, in turn, makes the whole response curve much more amenable to linearization.

There are many other instances where software linearization techniques will enhance the accuracy of the measuring system and at the same time allow simpler and cheaper components to be used.

9.3 Polynomial linearization

The most common method of linearizing the output of a measuring system is to apply a mathematical function known as a polynomial. The polynomial function is usually derived by the least-squares technique.

Polynomial least-squares fitting

We have already seen that the technique of least-squares fitting can generate coefficients of a straight-line equation representing the response of a linear measuring system. The least-squares method can be applied to fit other equations to *non-linear* response curves. The principle of the method is the same although, because we are now dealing with more complex curves and mathematical functions, the details of the implementation are slightly more involved.

A polynomial is a simple equation consisting of the sum of several separate terms. For the purposes of sensor calibration we can define a polynomial as an equation which describes how a dynamic variable, y, such as temperature or pressure (which we intend to measure) varies in relation to the corresponding transduced signal, x (e.g. voltage output or ADC reading). Each term consists of some known function of x multiplied by an unknown coefficient.

If we can determine the coefficients of a polynomial function that closely fits a set of measured calibration reference points, it is then possible to accurately calculate a value for the physical variable, y, from any ADC reading, x.

Formulating the best-fit condition

This section outlines the way in which the conditions for the best fit between the polynomial and data points can be derived. A more detailed account of this technique can be found in many texts on numerical analysis and, in particular, in the books by Miller (1993) and Press *et al.* (1992).

Suppose we have determined a set of calibration reference points (x_1, y_1), (x_2, y_2) to (x_n, y_n), where the x_i values represent the ADC reading (or corresponding transduced voltage reading) and y_i are values of the equivalent 'real-world' physical variable (e.g. temperature, displacement etc.).

In certain circumstances, some of the y_i values will be more accurate than others and it is advantageous to pay proportionally more regard to the most accurate points. To this end, the data points can be individually weighted by a factor w_i. This is usually set equal to the inverse of the square of the known error for each point. The w_i terms have been included in the following account of the least-squares method, but, in most circumstances, each reference point is measured in the same way, with the same equipment, and the accuracy (and therefore weight) of each point will usually be identical. In this case all of the w_i values can effectively be ignored by setting them to unity.

The polynomial which we wish to fit to the (x_i, y_i) calibration points is:

$$y'(x) = a_0 g_0(x) + a_1 g_1(x) + a_2 g_2(x) + \cdots + a_m g_m(x) = \sum_{k=0}^{k=m} a_k g_k(x) \tag{9.4}$$

There may be any number of terms in the polynomial. In this equation there are $m + 1$ terms, but it is usual for between 2 and 15 terms to be used. The number m is known as the order of the polynomial. As m increases, the polynomial is able to provide a more accurate fit to the calibration reference points. There are, however, practical limitations on m which we will consider shortly. In this equation the a_k values are a set of constant coefficients and $g_k(x)$ represents some function of x, which will remain unspecified for the moment.

At any given order, m, the polynomial will usually not fit the data points exactly. The deviation, δ_i, of each y_i reading from the fitted polynomial $y'(x_i)$ value is

$$\delta_i = \sum_{k=0}^{k=m} [a_k g_k(x_i)] - y_i \tag{9.5}$$

The principle of the least-squares method is to choose the a_k coefficients of the polynomial so as to minimize the sum of the squares of all δ_i values (known as the residue). Taking into account the weights of the individual points the residue, R, is given by

$$R = \sum_{i=1}^{i=n} w_i \delta_i^2 \tag{9.6}$$

The condition under which the polynomial will most closely fit the calibration reference points is obtained when the partial derivatives of R with respect to each a_k coefficient are all zero. This statement actually represents $m + 1$ separate conditions which must all be satisfied simultaneously for the best fit. Space precludes a full derivation here, but with a little algebra it is a simple matter to find that each of these conditions reduces to:

$$\sum_{i=1}^{i=n} w_i g_j(x_i) \sum_{k=0}^{k=m} (a_k g_k(x_i)) - y_i = 0 \tag{9.7}$$

As the best-fit is described by a set of $m + 1$ equations of this type (for $j = 0$ to m) we can represent them in matrix form as follows.

$$\begin{matrix} \alpha_{0,0} & \alpha_{1,0} & \alpha_{2,0} & \cdots & \alpha_{m,0} \\ \alpha_{0,1} & \alpha_{1,1} & \alpha_{2,1} & \cdots & \alpha_{m,1} \\ \alpha_{0,2} & \alpha_{1,2} & \alpha_{2,2} & \cdots & \alpha_{m,2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{0,m} & \alpha_{1,m} & \alpha_{2,m} & \cdots & \alpha_{m,m} \end{matrix} \cdot \begin{matrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_m \end{matrix} = \begin{matrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \cdots \\ \beta_m \end{matrix} \tag{9.8}$$

where

$$\alpha_{kj} = \sum_{i=1}^{i=n} w_i g_k(x_i) g_j(x_i)$$

$$\beta_j = \sum_{i=1}^{i=n} w_i g_j(x_i) y_i$$

Solving the best-fit equations

The matrix equation (9.8) represents a set of simultaneous equations which we need to solve in order to determine the coefficients, a_j, of the polynomial. The simplest method for solving the equations is to use a technique known as Gaussian Elimination to manipulate the elements of the matrix and vector so that they can then be solved by simple back-substitution.

The objective of Gaussian Elimination is to modify the elements of the matrix so that each position below the major diagonal is zero. This may be achieved by reference to a series of so-called pivot elements which lie at each successive position along the major diagonal. For each pivot element, we eliminate the elements below the pivot position by a systematic series of scalar-multiplication and row-subtraction operations as illustrated by the following code fragment.

```
for (Row = Pivot + 1; Row <= M; Row++)
  {
  Temp = Matrix[Pivot][Row] / Matrix[Pivot][Pivot];
  for (Col = Pivot; Col <= M; Col++)
    Matrix[Col][Row] = Matrix[Col][Row] - Temp * Matrix[Col][Pivot];
  Vector[Row] = Vector[Row] - Temp * Vector[Pivot];
  }
```

The variable `M` represents the order of the polynomial. `Matrix` is a square array with indices from 0 to `M`. This algorithm is used in the `GaussElim()` function shown in Listing 9.2 later in this chapter. Once all of the elements have been eliminated from below the major diagonal, the matrix equation will have the following form. The new matrix and vector elements are identified by primes to denote that the Gaussian Elimination procedure has generated different numerical values from the original $\alpha_{k,j}$ and β_j elements.

$$\begin{matrix} \alpha'_{0,0} & \alpha'_{1,0} & \alpha'_{2,0} & \cdots & \alpha'_{m,0} \\ 0 & \alpha'_{1,1} & \alpha'_{2,1} & \cdots & \alpha'_{m,1} \\ 0 & 0 & \alpha'_{2,2} & \cdots & \alpha'_{m,2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & \alpha'_{m,m} \end{matrix} \cdot \begin{matrix} a_0 \\ a_1 \\ a_2 \\ \cdots \\ a_m \end{matrix} = \begin{matrix} \beta'_0 \\ \beta'_1 \\ \beta'_2 \\ \cdots \\ \beta'_m \end{matrix} \qquad (9.9)$$

The equations represented by each row of the matrix equation can now be easily solved by repeated back-substitution. Starting with the bottom row and moving on to each higher row in sequence we can calculate a_m then a_{m-1} then a_{m-2} etc. as follows

$$a_m = \frac{\beta'_m}{\alpha'_{m,m}} \quad \text{then} \quad a_{m-1} = \frac{\beta'_{m-1} - a_m\alpha'_{m,m-1}}{\alpha'_{m-1,m-1}} \quad \text{etc.} \qquad (9.10)$$

In general we have the following iterative relation which is coded as a simple algorithm at the end of the `GaussElim()` function in Listing 9.2.

$$a_j = \frac{\beta'_j - \sum\limits_{l=j+1}^{l=m} a_l\alpha'_{l,j}}{\alpha_{j,j}} \qquad (9.11)$$

The curve fitting procedure would not usually need to be performed in real time and so the computation time required to determine coefficients by this method will not normally be of great importance. A 15th order polynomial fit can be carried out in several hundred milliseconds on an average 33 MHz 80486 machine equipped with a numeric processing unit, but will take considerably longer (up to a few seconds, depending upon the machine) if a coprocessor is not used. The total calculation time increases roughly in proportion to cube of the matrix size.

A number of other methods can be used to solve the matrix equation. These may be preferable if Gaussian Elimination fails to provide a solution because the coefficient matrix is singular, or if rounding errors become problematic. A discussion of these techniques is beyond the scope of this book. Press *et al.* (1992) provide a detailed description of curve fitting methods together with a comprehensive discussion of their relative advantages and drawbacks.

Numerical accuracy and ill-conditioned matrices

All computer-based numerical calculations are limited by the finite accuracy of the coprocessor or floating-point library used. Gaussian Elimination involves many repeated multiplications, divisions and subtractions. Consequently rounding errors can begin to accumulate, particularly with higher order polynomials. While single precision arithmetic is suitable for many of the calculations that we have to deal with in data-acquisition applications, it does not usually provide sufficient accuracy for polynomial linearization. When undertaking this type of calculation, it is generally beneficial to use floating-point data types with the greatest possible degree of precision. The examples presented in this chapter use C's `long double` data type, which is the largest type supported by the 80x87 family of numeric coprocessors.

Even when using the `long double` data type, rounding errors can become significant when undertaking Gaussian Elimination. For this reason it is generally inadvisable to attempt this for polynomials of greater than about 15th order. In some cases, rounding errors may also be important with lower order polynomials. If the magnitudes of the pivot elements vary greatly along the major diagonal, the process of Gaussian Elimination may cause rounding errors to build up to a significant level and it will then be impossible to calculate accurate values for the polynomial coefficients. The accuracy of the Gaussian Elimination method can be improved by first swapping the rows of the matrix equation so that the element in the pivot row with the largest absolute magnitude is placed in the pivot position on the

major diagonal. This minimizes the difference between the various pivot elements and helps to reduce the effect of rounding errors on the calculations.

If one of the pivot elements is zero the matrix equation cannot be solved by Gaussian Elimination. If one or more of the pivot elements are very close to zero the solution of the matrix equation may generate very large polynomial coefficients. Then when we subsequently evaluate the polynomial the greatest part of these coefficients tend to cancel each other out, leaving only a small remainder which contributes to the actual evaluation. This is obviously quite susceptible to numerical rounding errors.

The combination of elements in the matrix might be such that rounding errors in some of the operations performed during the elimination procedure become comparable with the true result of the operation. In this case the matrix is said to be ill-conditioned and the solution process may yield inaccurate coefficients.

It is usually advisable to check for ill-conditioned matrices by examining the pivot elements along the major diagonal to ensure that they do not differ by very many orders of magnitude. Obviously, if higher precision data types are used for calculation and storage of results (e.g. extended or double precision rather than single precision), it is possible to accommodate a greater range of values along the major diagonal.

It is also possible to detect the effect of ill-conditioned matrices and rounding errors after the fit has been performed. This can be achieved by carrying out conformance checks, as described in the next subsection, for a range of polynomial orders. This is not a foolproof technique, but in general, the root-mean-square deviation between the calibration reference points and the fitted polynomial will tend to increase with increasing order once rounding errors become significant.

Accuracy of the fitted curve

In the absence of any appreciable rounding errors, the accuracy with which the polynomial will model the measuring system's response curve will be determined by two factors: the magnitude of any random or systematic measurement errors in the calibration reference points and the 'flexibility' of the polynomial.

Although the effect of random errors can be offset to some extent by taking a larger number of calibration measurements, any systematic errors cannot generally be determined or corrected during linearization and so must be eliminated at source. There are many possible sources of random error. Electrical noise can be a problem with low voltage signals such as those generated

by thermocouples. There are also often difficulties in setting the measurand to a precise enough value, especially where the sensor is an integral part of a larger system and has to be calibrated *in situ*. Whatever the source of a random error, it generally introduces some discrepancy between the true response curve and the measured calibration reference points.

A second source of inaccuracy might arise where the polynomial is not flexible enough to fit response curves with rapidly changing gradients or higher derivatives. Better fits can usually be achieved by using high order polynomials, but, as mentioned previously, rounding errors can become problematic if very high orders are used.

Whenever a polynomial is fitted to a set of calibration reference points it is essential to obtain some measure of the accuracy of the fit. We can determine the uncertainties in the coefficients if we solve the best-fit equation (9.8) by the technique of Gauss–Jordan Elimination. As part of the Gauss–Jordan Elimination procedure we determine the inverse of the coefficient matrix and this can be used to calculate the uncertainties in the coefficients. The Gauss–Jordan method is somewhat more involved than Gaussian Elimination and, apart from providing an easy means of calculating the coefficient errors, has no other advantage. This method is discussed by Press *et al.* (1992) and will not be described here.

A simpler way of estimating the accuracy of the fit is to calculate the conformance between the fitted curve and each calibration reference point. We simply evaluate the polynomial $y'(x_i)$ for each x_i value in turn and then determine the deviation of the corresponding measured y_i value from the polynomial (see Equation 9.5). This is illustrated by the following code fragment.

```
SumDevnSq = 0;
WorstDev  = 0;
for (I = 0; I < NumPoints; I++)
 {
 Deviation = Y[I] - PolynomialValue(Order,X[I]);
 if (fabs(Deviation) > fabs(WorstDev)) WorstDev = Deviation;
 SumDevnSq = SumDevnSq + (Deviation * Deviation);
 }
RMSDev = sqrt(SumDevnSq / (NumPoints-2));
```

In this example, the polynomial is evaluated for the `I`th data point by calling the `PolynomialValue()` function (which will, of course, vary depending upon the functional form of the polynomial). A function of this type for evaluating a power-series polynomial is included in Listing 9.2 later in this chapter.

It is important not to rely too heavily on the conformance values calculated in this way. They show only how closely the polynomial fits the calibration reference points and do not indicate how the polynomial might vary from the true response curve between the points. It is advisable to check the accuracy of the polynomial at a number of points in between the original calibration reference points.

Choosing the optimum order

In general the higher the order of the polynomial the more closely it will fit the calibration reference points. One might be tempted always to fit a very high order polynomial, but this has several disadvantages. First, high order polynomials take longer to evaluate and, as the evaluation process is likely to be carried out in real time, this can severely limit throughput. Second, rounding errors tend to be more problematic with higher order polynomials as already discussed. Finally, more calibration reference points are required in order to obtain a realistic approximation to the response curve.

For any polynomial fit, the number of calibration reference points used must be greater than $m+1$, where m is the order of the polynomial. If this rule is broken, by choosing an order which is too high, the fitting procedure will not provide accurate coefficients and the polynomial will tend to deviate from a reasonably smooth curve between adjacent data points. In order to obtain a smooth fit to the response curve it is always advisable to use as many calibration reference points as possible, and the lowest order of polynomial consistent with achieving the required accuracy. As the order of the fit is increased, the rms deviation between the fitted polynomial and the reference points will normally tend to decrease and then level out as shown in Figure 9.2.

The shape of the graph will, of course, vary for different data sets, but the same general trends will usually be obtained. In this example, there is little to be gained by using an order greater than about 11 or 12. At higher orders rounding errors may begin to come into play causing the rms deviation to rise irregularly. If the requirements of an application are such that a lower degree of accuracy would be acceptable, it is generally preferable to employ a lower order polynomial, for the reasons mentioned above.

Linearization with power-series polynomials

So far, in the discussion of the least-squares technique, the form of the $g_k(x)$ function has not been specified. In fact, it may be almost any continuous function of x such as $\sin(x)$, $\ln(x)$ etc. For correcting the response of a non-linear sensor it is usual to use a

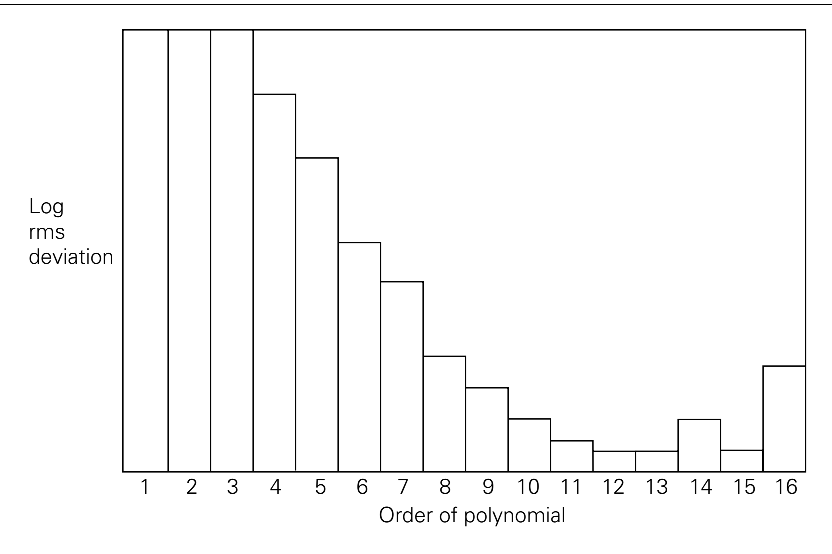


Figure 9.2 *Typical rms deviation vs. order for a power-series polynomial fit*

power-series polynomial where each successive term is proportional to an increasing power of x. For a power-series polynomial, the elements of the matrix and vector in Equation 9.11 become

$$\alpha_{kj} = \sum_{i=1}^{i=n} w_i x_i^k x_i^j \quad \text{and} \quad \beta_j = \sum_{i=1}^{i=n} w_i x_i^j y_i \tag{9.12}$$

By setting all weights to unity, substituting these elements into the matrix equation for a first order polynomial and then solving for a_0 and a_1 we can arrive at Equations 9.3 for the parameters of a straight line which were presented in the *Multiple-point prime calibration* section. (Note that the following substitutions must be made: $a_1 = s$; $a_0 = h$.)

Power-series polynomials are a special case of the generalized polynomial function fit and are useful for correcting a variety of non-linear response curves. They are, perhaps, most often employed for linearizing thermocouple signals but they can also be used with a number of other types of non-linear sensor. The resistance vs. temperature characteristic of a platinum RTD, for example, can be linearized with a second order power-series polynomial (Johnson, 1988), but for higher accuracy or wider temperature ranges a third or fourth order polynomial should be used. Higher (typically 8th to 14th) order polynomials are required to linearize thermocouple signals, as the response curves of these devices tend to be quite non-linear. Power-series polynomials are most effective where the

response curve deviates smoothly and gradually from linearity (as is usually the case with thermocouple signals), but they normally provide a poorer fit to curves that contain sudden steps, bumps or discontinuities.

Non-linearities often stem from the design of the transducer and its associated signal conditioning circuits. However, power-series polynomials can also be used in cases where other sources of non-linearity are present. For example, the mechanical design of a measuring system might require a displacement transducer such as an LVDT to be operated via a series of levers in order to indirectly measure the rotational angle of some component. In this case, although the response of the LVDT and signal conditioning circuits are essentially linear, the transducer's output will have a non-linear relation to the quantity of interest. Systems such as this often exhibit smooth deviations from linearity and can usually be linearized with a power-series polynomial.

Fitting a power-series polynomial

To fit a polynomial of any chosen order to a set of calibration reference points, it is first necessary to construct a matrix equation with the appropriate terms (as defined by Equations 9.12). The matrix should be simplified using the Gaussian Elimination technique described in the previous section and the coefficients may then be calculated by back-substitution.

Listing 9.2 shows how a power-series polynomial can be fitted to an unweighted set of calibration reference points. As each point is assumed to have been determined to the same degree of precision, all weights in Equations 9.12 can be set to unity. If required, weights could easily be incorporated into the code by modifying the first block of lines in the `PolynomialLSF()` function.

The code in this listing will automatically attempt to fit polynomials of all orders up to a maximum order which is limited by either the matrix size or the number of available calibration points. The present example accommodates a 16×16 matrix which is sufficient for a 15th order polynomial. If necessary, the size of the matrix can be increased by modifying the `#define N` line. Bear in mind, however, that if larger matrices and polynomials are used, rounding errors may become problematic. As mentioned in the previous section, polynomial fits should not be attempted for orders greater than $n - 2$, where n represents the number of calibration reference points. The code will, therefore, not attempt to fit a polynomial if there are insufficient points available.

The (x_i, y_i) data for the fit are made available to the fitting functions in the global `x` and `y` arrays. The results of the fitting are

Listing 9.2 *Fitting a power-series polynomial to a set of calibration data points*

```
#include <math.h>

#define True    1
#define False   0
#define MaxNP   500                /* Maximum number of data points for fit */
#define N       16    /* No. of terms. 16 accommodates 15th order polynomial */

struct OrderRec
  {
  long double Coef[N];                            /* Polynomial coefficients */
  double      RMSDev;      /* RMS deviation of polynomial from Y data points */
  double      WorstDev;  /* Worst deviation of polynomial from Y data points */
  };

struct PolyFitResults
  {
  unsigned char   MaxOrder;             /* Highest order of polynomial to fit */
  struct OrderRec ForOrder[N];       /* Polynomial parameters for each order */
  };

struct PolyFitResults PResults;
long double       Matrix[N][N];                   /* Matrix in equation 10.11 */
long double       Vector[N];                      /* Vector in equation 10.11 */
unsigned int      NumPoints;                   /* Number of (X,Y) data points */
double            X[MaxNP];                                     /* X data */
double            Y[MaxNP];                                     /* Y data */


long double Power(long double X, unsigned char P)
/* Calculates X raised to the power P */
{
unsigned char I;
long double   R;

R = 1;
if (P > 0)
  for (I = 1; I <= P; I++)
    R = R * X;
return(R);
}


void GaussElim(unsigned char M, long double Solution[N], unsigned char *Err)
/* Solves the matrix equation contained in the global Matrix and Vector arrays
   by Gaussian Elimination and back-substitution. Returns the solution vector
   in the Solution array.
*/
{
signed char Pivot;
signed char JForMaxPivot;
signed char J;
signed char K;
signed char L;
long double   Temp;
long double   SumOfKnownTerms;
```

Listing 9.2 *(continued)*

```
*Err = False;

/* Manipulate the matrix to produce zeros below the major diagonal */
for (Pivot = 0; Pivot <= M; Pivot++)
  {
  /* Find row with the largest value in the Pivot column */
  JForMaxPivot = Pivot;
  if (Pivot < M)
    for (J = Pivot + 1; J <= M; J++)
      if (fabsl(Matrix[Pivot][J]) > fabsl(Matrix[Pivot][JForMaxPivot]))
        JForMaxPivot = J;

  /* Swap rows of matrix and vector so that the largest matrix */
  /* element is in the Pivot row (ie. falls on the major diagonal) */
  if (JForMaxPivot != Pivot)
    {
    /* Swap matrix elements. Note that elements with K < Pivot are all */
    /* zero at this stage and may be ignored. */
    for (K = Pivot; K <= M; K++)
      {
      Temp                  = Matrix[K][Pivot];
      Matrix[K][Pivot]       = Matrix[K][JForMaxPivot];
      Matrix[K][JForMaxPivot] = Temp;
      }

    /* Swap vector "rows" (ie. elements) */
    Temp                 = Vector[Pivot];
    Vector[Pivot]        = Vector[JForMaxPivot];
    Vector[JForMaxPivot] = Temp;
    }

  if (Matrix[Pivot][Pivot] == 0)
        *Err = True;
    else {
        /* Eliminate variables in matrix to produce zeros in all */
        /* elements below the pivot element */

        for (J = Pivot + 1; J <= M; J++)
          {
          Temp = Matrix[Pivot][J] / Matrix[Pivot][Pivot];
          for (K = Pivot; K <= M; K++)
            Matrix[K][J] = Matrix[K][J] - Temp * Matrix[K][Pivot];
          Vector[J] = Vector[J] - Temp * Vector[Pivot];
          }
        }
  }

/* Solve the matrix equations by backsubstitution, starting with */
/* the bottom row of the matrix */
if (!(*Err))
  {
  if (Matrix[M][M] == 0)
        *Err = True;
    else {
        for (J = M; J >= 0; J--)
          {
          SumOfKnownTerms = 0;
```

Listing 9.2 *(continued)*

```
          if (J < M)
            for (L = J + 1; L <= M; L++)
              SumOfKnownTerms = SumOfKnownTerms + Matrix[L][J] * Solution[L];
          Solution[J] = (Vector[J] - SumOfKnownTerms) / Matrix[J][J];
          }
        }
  }
}



void PolynomialLSF(unsigned char Order, unsigned char *Err)
/* Performs a polynomial fit on the X, Y data arrays of the specified Order
   and stores the results in the global PResults structure.
*/
{
long double   MatrixElement[2 * (N - 1) + 1];           /* Temporary storage */
unsigned char KPlusJ;                               /* Index of matrix elements */
unsigned char K;                                   /* Index of coefficients */
unsigned char J;                         /* Index of equation / vector elements */
unsigned int  I;                                     /* Index of data points */

/* Sum data points into Vector and MatrixElement array. MatrixElement is */
/* used for temporary storage of elements so that it is not necessary to */
/* duplicate the calculation of identical terms */
for (KPlusJ = 0; KPlusJ <= (2 * Order); KPlusJ++) MatrixElement[KPlusJ] = 0;
for (J = 0; J <= Order; J++) Vector[J] = 0;
for (I = 0; I < NumPoints; I++)
  {
  for (KPlusJ = 0; KPlusJ <= (2 * Order); KPlusJ++)
    MatrixElement[KPlusJ] = MatrixElement[KPlusJ] + Power(X[I],KPlusJ);
  for (J = 0; J <= Order; J++)
    Vector[J] = Vector[J] + (Y[I] * Power(X[I],J));
  }

/* Copy matrix elements to Matrix */
for (J = 0; J <= Order; J++)
  for (K = 0; K <= Order; K++)
    Matrix[K][J] = MatrixElement[K+J];

/* Solve matrix equation by Gaussian Elimination and backsubstitution. */
/* Store the solution vector in the Results.ForOrder[Order].Coef array. */
GaussElim(Order,PResults.ForOrder[Order].Coef,Err);
}



long double PolynomialValue(unsigned char Order, double X)
/* Evaluates the polynomial contained in the global PResults structure.
   Returns the value of the polynomial of the specified order at the
   specified value of X.
*/
{
signed char K;
long double P;
```

Listing 9.2 *(continued)*

```
P = PResults.ForOrder[Order].Coef[Order];
for (K = Order - 1; K >= 0; K--)
  P = P * X + PResults.ForOrder[Order].Coef[K];
return P;
}



void CalculateDeviation(unsigned char Order)
/* Calculates the root-mean-square and worst deviations of all Y values from
   the fitted polynomial.
*/
{
unsigned int I;
double       Deviation;
double       SumDevnSq;

SumDevnSq = 0;
PResults.ForOrder[Order].WorstDev  = 0;
for (I = 0; I < NumPoints; I++)
  {
  Deviation = Y[I] - PolynomialValue(Order,X[I]);
  if (fabs(Deviation) > fabs(PResults.ForOrder[Order].WorstDev))
    PResults.ForOrder[Order].WorstDev = Deviation;
  SumDevnSq = SumDevnSq + (Deviation * Deviation);
  }
PResults.ForOrder[Order].RMSDev = sqrt(SumDevnSq / (NumPoints-2));
}



void PolynomialFitForAllOrders(unsigned char *Err)
/* Performs a polynomial fit for all orders up to a maximum determined by the
   number of data points and the dimensions of the Matrix and Vector arrays.
*/
{
unsigned char Order;

*Err = False;
if (NumPoints > N)
       PResults.MaxOrder = N - 1;
  else PResults.MaxOrder = NumPoints - 2;
for (Order = 1; Order <= PResults.MaxOrder; Order++)
  {
  if (!(*Err))
    {
    PolynomialLSF(Order,Err);
    if (!(*Err)) CalculateDeviation(Order);
    }
  }
}
```

stored in the global `PResults` structure (of type `PolyFitResults`). The `PolynomialFitForAllOrders()` function performs a polynomial fit to the same data over a range of orders by calling the `PolynomialLSF()` function once for each order. This constructs the matrix and vector defined in Equation 9.11 using the appropriate power-series polynomial terms and then calls the `GaussElim()` function to solve the matrix equation. After each fit has been performed the `CalculateDeviation()` function determines the rms and worst deviation of the (x_i, y_i) points from the fitted curve.

All of the fitting calculations employ C's 80-bit `long double` floating-point data type. This is the same as Pascal's `extended` type and corresponds to the Intel 80x87 coprocessor's Temporary Real data type. These provide 19 to 20 significant digits over a range of about 3.4×10^{-4932} to $1.1 \times 10^{+4932}$.

The listing incorporates two functions that are actually included in some standard C libraries. Calls to the `Power()` function can be replaced by calls to the C `powl()` function if it is supported in your library. The function has been included here for the benefit of readers who wish to translate the code into languages such as Pascal, which might not have a comparable procedure. Users of Borland C++ or Turbo C/C++ may wish to replace the `PolynomialValue()` function with the `poly()` or `polyl()` library functions. However, these are not defined in ANSI C and are not supported in all implementations of the language.

Evaluating a power-series polynomial

In order to calculate the rms and worst deviation, it is necessary for the code to evaluate the fitted polynomial for each of the x_i values. The most obvious way to do this would have been to calculate each term individually and to sum them as follows.

```
PolyValue = 0;
for (K = 0; K <= Order; K++)
  PolyValue = PolyValue + Coef[K]*Power(X[I],K);
```

However, this requires x_i^k to be evaluated for each term, which results in many multiplication operations being performed unnecessarily by the `Power()` function. The following algorithm is much more efficient and requires only `Order` + 1 multiplications to be performed. Note that the index `K` is, in this case, a `signed char`.

```
PolyValue = Coef[Order];
for (K = Order-1; K >= 0; K--)
  PolyValue = PolyValue * X[I] + Coef[K];
```

For a 15th order polynomial the first method requires 121 separate multiply operations while only 16 are needed in the more efficient second method. The second method minimizes the effect of rounding errors and will often make a significant improvement to throughput.

Polynomials in other functions

A power-series polynomial can be useful where the functional form of a response curve is unknown or difficult to determine. However, the response of some measuring systems might clearly follow a combination of simple mathematical functions (sin, cos, log etc.) and in such cases it is likely that a low order polynomial in the appropriate function will provide a more accurate fit than a high order power-series polynomial.

Thermistors, for example, exhibit a resistance (R) vs. temperature (T) characteristic in which the inverse of the temperature is proportional to a polynomial in ln R (see Tompkins and Webster, 1988):

$$T^{-1} = a_0 + a_1 \ln R + a_3 (\ln R)^3 \tag{9.13}$$

A response curve based on a simple mathematical function might also arise where the non-linearity is introduced by the geometry of the measuring system. One example is that of level measurement using a float and linkage as shown in Figure 9.3.

The float moves up and down as the level of liquid in the tank changes and the resulting motion (i.e. angle a) of the mechanical

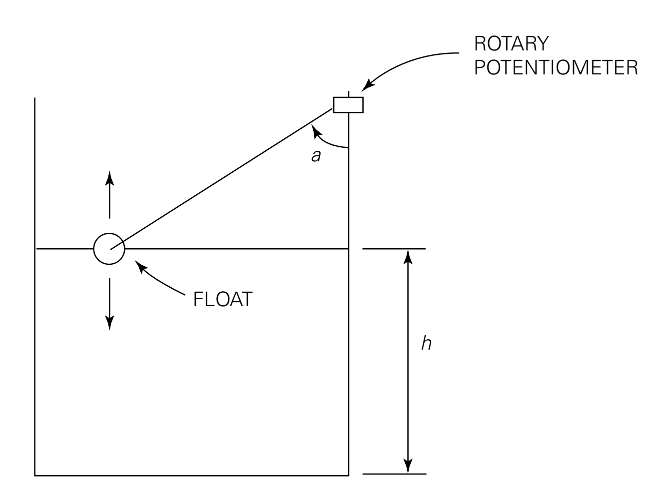


Figure 9.3 *Measurement of fluid level using a float linked to a rotary potentiometer*

link is sensed by a rotary potentiometric transducer. The output of the potentiometer is assumed to be proportional to a, and the level, h, of liquid in the tank will be approximately proportional to $\cos(a)$.

The best approach might initially seem to be to scale the output of the potentiometer to obtain the value of a and then apply the simple $\cos(a)$ relationship in order to calculate h. This might indeed be accurate enough, but we should remember that there may be other factors which affect the actual relationship between h and the potentiometer's output. For example, the float might sit at a slightly different level in the liquid depending upon the angle a and this will introduce a small deviation from the ideal cosinusoidal response curve. Deviations such as this are usually best accounted for by performing a prime calibration and then linearizing the resulting calibration points with the appropriate form of polynomial.

The polynomial fitting routine in Listing 9.2 can easily be modified to accommodate functions other than powers of x. There are only two changes which usually need to be made. The first is that the `PolynomialLSF()` function should be adapted to calculate the matrix elements from the appropriate $g_k(x)$ functions. The other modification required is in the three lines of code in the `PolynomialValue()` function which evaluates the polynomial at specific points on the response curve.

9.4 Interpolation between points in a look-up table

Suppose that a number of calibration points, (x_1, y_1), (x_2, y_2) to (x_n, y_n), have been calculated, or measured using the prime calibration techniques discussed previously. If there are sufficient points available, it is possible to store them in a look-up table and to use this table to directly convert the ADC reading into the corresponding 'real-world' value. In cases where a low resolution ADC is in use it might be feasible to construct a table containing one entry for each possible ADC reading. This, however, requires a large amount of system memory, particularly if there are several ADC channels, and it is normally only practicable to store more widely separated reference points. In order to avoid having to round down (or up) to the nearest tabulated point it is usual to adopt some method of interpolating between two or more neighbouring points.

Sorting the table of calibration points

The first step in finding the required interpolate is to determine which of the calibration points the interpolation should be based on.

Two (or more) points with *x* values spanning the interpolation point are required, and the software must undertake a search for these points. In order to maximize the efficiency of the search routine (which often has to be executed in real time), the data should previously have been ordered such that the *x* values of each point increase (or decrease) monotonically through the table.

The points may already be correctly ordered if they have been entered from a published table or read in accordance with a strict calibration algorithm. However, this may not always be the case. It is prudent to provide the operator with as much flexibility as possible in performing a prime calibration and this may mean relaxing any constraints on the order in which the calibration points are entered or measured. In this case it is likely that the look-up table will initially contain a randomly ordered set of measurements which will have to be rearranged into a monotonically increasing or decreasing sequence.

One of the most efficient ways of sorting a large number (up to about 1000) of disordered data points is shown in Listing 9.3. This is based on the Shell–Metzner sorting algorithm (Knuth, 1973, Press *et al.*, 1992) and arranges any randomly ordered table of (*x*, *y*) points into ascending *x* order.

The `ShellSort()` function works by comparing pairs of *x* values during a number of passes through the data. In each pass the compared values are separated by `DeltaI` array locations and `DeltaI` is halved on each successive pass. The first few passes through the data introduce a degree of order over a large scale and subsequent passes reorder the data on continually smaller and smaller scales.

This might seem to be an unnecessarily complicated method of sorting, but it is considerably more efficient than some of the simpler algorithms (such as the well-known Search-and-Insert or Bubble Sort routines), particularly if the data set contains more than about 30 to 40 points. The time required to execute the `ShellSort()` algorithm increases in proportion to `NumPoints` to the power of 1.5 or less, while the execution time for a Bubble Sort increases with `NumPoints` squared. However, if there are only a small number of calibration points (less than about 20 to 30) to be sorted the simpler `BubbleSort()` routine shown in Listing 9.4 will generally execute faster than `ShellSort()`.

The C language includes a `qsort()` function which can be used to sort an array of data according to the well-known Quick Sort algorithm. This algorithm is ideal when dealing with large quantities of data (typically >1000 items), but for smaller arrays of calibration points, a well-coded implementation of the Shell–Metzner technique tends to be more efficient.

Listing 9.3 *Sorting routine based on the Shell–Metzner (Shell Sort) algorithm for use with up to approximately 1000 data points*

```
#define True    1
#define False   0
#define MaxNP   500       /* Maximum number of data points in lookup table */

void ShellSort(unsigned int NumPoints, double X[MaxNP], double Y[MaxNP])
/* Sorts the X and Y arrays according to the Shell-Metzner algorithm so that
   the contents of the X array are placed in ascending numeric order. The
   corresponding elements of the Y array are also interchanged to preserve the
   relationship between the two arrays.
*/
{
unsigned int  DeltaI;                /* Separation between compared elements */
unsigned char PointsOrdered;    /* True indicates points ordered on each pass */
unsigned int  NumPairsToCheck; /* No. of point pairs to compare on each pass */
unsigned int  I0,I;                   /* Indices for search through arrays */
double        Temp;                /* Temporary storage for swapping points */

if (NumPoints > 1)
  {
  DeltaI = NumPoints;

  do
    {
    DeltaI = DeltaI / 2;

    /* Compare pairs of points separated by DeltaI */
    do
      {
      PointsOrdered   = True;
      NumPairsToCheck = NumPoints - DeltaI;
      for (I0 = 0; I0 < NumPairsToCheck; I0++);
        {
        I = I0 + DeltaI;
        if (X[I0] > X[I])
          {
          /* Swap elements of X array */
          Temp  = X[I];
          X[I]  = X[I0];
          X[I0] = Temp;

          /* Swap elements of Y array */
          Temp  = Y[I];
          Y[I]  = Y[I0];
          Y[I0] = Temp;

          PointsOrdered = False;    /* Not yet ordered so do same pass again */
          }
        }
      }
    while (!PointsOrdered);
    }
  while (DeltaI != 1);
  }
}
```

Listing 9.4 *Bubble Sort routine for use with fewer than approximately 20 to 30 data points*

```
#define MaxNP   500        /* Maximum number of data points in lookup table */

void BubbleSort(unsigned int NumPoints, double X[MaxNP], double Y[MaxNP])
/* Sorts the X and Y arrays according to the Bubble Sort algorithm so that
   the contents of the X array are placed in ascending numeric order. The
   corresponding elements of the Y array are also interchanged to preserve the
   relationship between the two arrays.
*/
{
unsigned int I;
unsigned int I0;
double       Temp;

for (I0 = 0; I0 < NumPoints - 1; I0++)
  {
  for (I = I0 + 1; I < NumPoints; I++)
    {
    if (X[I0] > X[I])
      {
      /* Swap elements of X array */
      Temp  = X[I];
      X[I]  = X[I0];
      X[I0] = Temp;

      /* Swap elements of Y array */
      Temp  = Y[I];
      Y[I]  = Y[I0];
      Y[I0] = Temp;
      }
    }
  }
}
```

The Bubble Sort algorithm is notoriously inefficient and should be used only if the number of data points is small. Do not be tempted to use a routine based on the Bubble Sort method with more than about 20 to 30 points. It becomes very slow if large tables of data have to be sorted and, in these cases, it is worth the slight extra coding effort to replace it with the Shell Sort routine.

There are many other types of sorting algorithm. Most of these are, however, designed specially for sorting very large quantities of data and there is usually no significant advantage to be gained by using them in preference to the `ShellSort()` function. See Press *et al.* (1992) and Knuth (1973) for more detailed discussions of this topic.

The sorting process should, of course, be performed immediately after the calibration reference points have been entered or measured. It should not be deferred until run time as this is likely to place an unacceptable burden on the real-time operation of the software.

Searching the look-up table

In order to determine which calibration points will be used for the interpolation, the software must search the previously ordered table. The most efficient searching routines tend to be based on bisection algorithms such as that identified by the `Bisection Search` comment in Listing 9.5. This routine searches through a portion of the table (defined by the indices `Upper` and `Lower`) by repeatedly halving it. It decides which portion of the table is to be bisected next by comparing the bisection point (`Bisect`) with the required interpolation point (`TargetX`). The bisection algorithm rapidly converges on the pair of data points with *x* values spanning `TargetX` and returns the lower of the *indices* of these two points. This is similar, in principle, to the successive-approximation technique employed in some analogue-to-digital converters.

Listing 9.5 *Delimit-and-bisect function for searching an ordered table*

```
#define True    1
#define False   0
#define MaxNP   500        /* Maximum number of data points in lookup table */

void Search(unsigned int NumEntries, double X[MaxNP], double TargetX,
            signed int *Index, unsigned char *Err)
/* Searches the ascending table of X values by bracketing and then bisection.
   This procedure will not accommodate descending tables. NumEntries specifies
   the number of entries in the X array and should always be less than 32768.
   Bracketing starts at the entry specified by Index. The bisection search then
   returns the index of the entry such that X[Index] <= TargetX < X[Index+1].
   If Index is out the range 1 to NumEntries, the bisection search is performed
   over the whole table. If TargetX < X[1] or TargetX >= X[NumEntries], Err is
   set true.
*/
{
signed int   Span;
signed int   Upper;
signed int   Lower;
unsigned int Bisect;

if (X[0] > X[NumEntries-1])
       *Err = True;                                              /*Descending*/
  else *Err = ((TargetX < X[0]) || (TargetX >= X[NumEntries-1]));  /*Ascending*/

if (!*Err)
  {

  /* Define search limits */
  if ((*Index >= 0) && (*Index < NumEntries))
         {
         /* Adjust bracket interval to encompass TargetX */
         Span = 1;
         if (TargetX >= X[*Index])
```

Listing 9.5 *(continued)*

```
                {
                /* Adjust upwards */
                Lower = *Index;
                Upper = Lower + 1;
                while (TargetX >= X[Upper])
                  {
                  Span  = 2 * Span;
                  Lower = Upper;
                  Upper = Upper + Span;
                  if (Upper > NumEntries - 1) Upper = NumEntries - 1;
                  }
                }
           else {
                /* Adjust downwards */
                Upper = *Index;
                Lower = Upper - 1;
                while (TargetX < X[Lower])
                  {
                  Span  = 2 * Span;
                  Upper = Lower;
                  Lower = Lower - Span;
                  if (Lower < 0) Lower = 0;
                  }
                }
         }
    else {
         /* *Index is out of range so search the whole table */
         Lower = 0;
         Upper = NumEntries;
         }

  /* Bisection search */
  while ((Upper - Lower) > 1)
    {
    Bisect = (Upper + Lower) / 2;
    if (TargetX > X[Bisect])
           Lower = Bisect;
      else Upper = Bisect;
    }
  *Index = Lower;

  }
}
```

The total execution time of the bisection search algorithm increases roughly in proportion to $\log_2(n)$, where n is the number of points in the range of the table to be searched.

The bisection routine would work reasonably well if the `Upper` and `Lower` search limits were to be set to encompass the whole table, but this can often be improved by including code to define narrower search limits. The reason is that, in many data-acquisition applications, there is a degree of correlation between successive readings. If the signal changes slowly compared to the sampling

rate, each consecutive reading will be only slightly different from the previous one. The `Search()` function takes advantage of any such correlation by starting the search from the last interpolation point. It initially sets the search range so that it includes only the last interpolation point (x value) used and then continuously extends the range in the direction of the new interpolation point until the new point falls within the limits of the search range. The final range is then used to define the boundaries of the subsequent bisection search.

The `Search()` function uses the initial value of the `Index` parameter to fix the starting point of the range-adjustment process. The calling program should usually initialize `Index` before invoking `Search()` for the first time and it should subsequently ensure that `Index` retains its value between successive calls to `Search()`. It is, of course, possible to cause the searching process to begin at any other point in the table just by setting `Index` to the required value before calling the `Search()` function.

If successive readings are very close, the delimit-and-bisect strategy can be considerably more efficient than always performing the bisection search across the whole table. The improvement in efficiency is most noticeable in applications which use extensive calibration tables. However, if successive readings are totally unrelated, this method will take approximately twice as long (on average) to find the required interpolation point.

The `Search()` function will work only on tables in which the x values are arranged in ascending numerical order, but it can easily be adapted to accommodate descending tables.

Interpolation

There are many types of interpolating function – the nature of each application will dictate which function is most appropriate. The important point to bear in mind when selecting an interpolating function is that it must be representative of the true form of the response curve over the range of interpolation. The present discussion will be confined to simple polynomial interpolation which (provided that the tabulated points are close enough) is a suitable model for many different shapes of response curve.

Any n adjacent calibration points describe a unique polynomial of order $n - 1$ that can be used to interpolate to any other point within the range encompassed by the calibration points. Lagrange's equation describes the interpolating polynomial of order $n - 1$

passing through any n points, (x_1, y_1), $(x_2, y_2) \cdots (x_n, y_n)$:

$$P(x) = \frac{(x - x_2)(x - x_3)\cdots(x - x_n)}{(x_1 - x_2)(x_1 - x_3)\cdots(x_1 - x_n)} y_1 + \frac{(x - x_1)(x - x_3)\cdots(x - x_n)}{(x_2 - x_1)(x_2 - x_3)\cdots(x_2 - x_n)} y_2 + \cdots + \frac{(x - x_1)(x - x_2)\cdots(x - x_{n-1})}{(x_n - x_1)(x_n - x_2)\cdots(x_n - x_{n-1})} y_n \quad (9.14)$$

The Lagrange polynomial can be evaluated at any point, x_i, where $1 \leq i \leq n$, in order to provide an estimate of the true response function $y(x_i)$.

The interpolating polynomial should not be confused with the best-fit polynomial determined by the least-squares technique. The $(n - 1)$th order interpolating polynomial passes *precisely* through the n reference points; the best-fit polynomial represents the closest approximation that can be made to the reference points using a polynomial of a specified order. In general the order of the best-fit polynomial is considerable smaller than the number of data points.

It is usually not advisable to use a high (i.e. greater than about fourth or fifth) order interpolating polynomial either, unless there is a good reason to believe that it would accurately model the real response curve. High order polynomials can introduce an excessive degree of curvature. They also rely on reference points that are more distant from the required interpolation point and these are, of course, less representative of the required interpolate.

The other important drawback with high order polynomial interpolation is that it involves quite complex and time-consuming calculations. As the interpolation usually has to be performed in real time, we are generally restricted to using low order (i.e. linear or quadratic) polynomials. The total execution time can be reduced if the calibration reference points are equally spaced along the x axis. We can see from Lagrange's equation that, in this case, it would be possible to simplify the denominators of each term and thus to reduce the number of arithmetic operations involved in performing the interpolation.

In order to avoid compromising the accuracy of the calibration, it is necessary to ensure that sufficient calibration reference points are contained within the look-up table. The points should be more closely packed in regions of the response curve that have rapidly changing first derivatives.

If the points are close enough, we can use the following simple linear interpolation formula

$$y = \frac{(x - x_i)(y_{i+1} - y_i)}{(x_{i+1} - x_i)} + y_i \tag{9.15}$$

A number of other interpolation techniques exist and these may occasionally be useful under special circumstances. For a thorough discussion of this topic the reader is referred to the texts by Fröberg (1966) and Press *et al.* (1992).

9.5 Interpolation vs. power-series polynomials

Interpolation can in some circumstances provide a greater degree of accuracy than linearization schemes that are based on the best-fit polynomial. A 50-point look-up table will approximate the response of a type-T thermocouple to roughly the same degree of accuracy as a 12th order polynomial. It is relatively easy to increase the precision of a look-up table by including more points, but increasing the order of a linearizing polynomial can be less straightforward because of the effect of rounding errors.

Interpolation using a look-up table can also be somewhat faster than evaluating the best-fit polynomial, particularly if the PC is not equipped with a numeric co-processor. The speed advantage obtained with look-up tables will, of course, depend upon the number of points in the table and the order of the polynomial. The time required to evaluate a power-series polynomial increases in proportion to its order. Using the `Search()` function in Listing 9.5, the total search time required prior to performing an interpolation increases approximately in proportion to $\log_2(\eta)$ where η represents the average number of elements to be searched. As mentioned previously, if successive readings are correlated, η can be quite small. As a rough rule-of-thumb, if a numeric coprocessor is used, it takes about the same length of time to evaluate a 12th order polynomial as to search a 25-point table and then perform a linear interpolation. If a co-processor is not available, the balance will tend to shift in favour of the search-and-interpolate technique.

9.6 Interactive calibration programs

The users of a data-acquisition program will probably be familiar with the measurements that it will be required to make. Indeed, it is quite possible that the software will have been commissioned in

order to computerize some process that the operator has already been carrying out for a number of years. The calibration procedure is not generally related to the logic of the data-acquisition process and, consequently, the end user is probably less likely to understand the steps involved in calibration than any other part of the measuring system. Calibration often requires quite a high degree of operator involvement. Any mistakes will have the potential to introduce serious errors into the measuring system and may disrupt the system's control functions.

For these reasons, calibration can be one of the most problematic aspects of a data-acquisition system and it is worthwhile making the calibration software as efficient, informative and easy to use as possible. This benefits not only the end user but also the supplier in fewer maintenance call-outs and telephone queries.

From the programmer's point of view, the simplest calibration routines are those which require the user to calculate scaling factors, offsets or polynomial coefficients and to type in these values for subsequent storage in a data file. Clearly, this procedure can be quite error prone. A more satisfactory alternative it to produce an *interactive* calibration program which continuously displays the output from the sensor and, when commanded to do so, samples the ADC output and automatically calculates scaling factors or linearization parameters. This reduces the operator's job to simply adjusting the sensor input and/or the signal-conditioning (e.g. amplifier gain) and then selecting the appropriate menu options on the PC. Whatever method is chosen, it cannot be overemphasized that the calibration program should be as simple to use as possible, and should minimize the potential for operator errors.

The user interface

The computer should, as far as possible, oversee the sequence of events that occur during the calibration process. The software might, for example, require the transducer's zero offset to be measured first, and a second calibration reference point to be obtained at the transducer's full-scale setting. It is, however, advisable to provide the operator with the option to abandon the calibration procedure and to either restart the whole process or to restore the scaling factor and other calibration parameters to their original values.

The calibration program's display screen should be as clear and informative as possible. Large digital displays might be used to indicate the current scaled and unscaled sensor readings, while analogue bar charts can provide a more graphic representation. Different colours can be used for the scaled and unscaled displays

in order to enhance clarity. It is sometimes useful to change the colours of the displays whenever the input is scaled or linearized, thereby shifting the visual emphasis from one set of indicators to the other. Other useful facilities might include a noise-monitoring facility to ensure that the level of noise present will not compromise calibration accuracy.

The calibration software should be designed to trap operator errors wherever possible. It should, for example, detect when *obviously* incorrect inputs are applied to the transducer. Clear on-screen instructions, information panels and help screens are of considerable value. Diagrams or other pictorial representations of the positions or status of the various sensors can also be a useful aid to understanding the calibration process.

9.7 Practical issues

Calibration is usually a straightforward matter if easy access is available to the sensor and if it is possible to use the appropriate type of measuring jig or calibration reference device. In many situations, however, the transducer forms part of a larger system – perhaps part of a machine working on a production line – and in these cases the transducer may have to be calibrated *in situ*. This often introduces a number of practical difficulties into the calibration process. By designing the software to take account of these difficulties it is possible to greatly simplify the procedures involved in calibration. A few of the relevant considerations are described below.

Flexible calibration sequence

At its simplest, prime calibration involves the following steps:

1. Sample the output of the measuring system with zero input.
2. Sample the output of the measuring system at (or near to) full scale.
3. Calculate the offset and scaling factor from the two previous calibration points.

Each of these steps may be performed in response to specific inputs from the user (e.g. a key press, menu selection or mouse click). Obviously, three-point and multiple-point calibration schemes would require more than two reference points to be obtained, but the basic principle still applies.

It should be borne in mind that, in multi-channel systems, there may be a correlation between the readings obtained with two or

more of the sensors – i.e. the various sensors might actually measure different aspects of the same physical process or object. For example, consider a system which uses 100 LVDTs in a gauging jig to measure the displacement at different locations on the surface of some manufactured component. It might be difficult to *individually* set each transducer to its zero position and then to its full-scale position using a set of gauge blocks. A more practical method would be to place two dummy components, or spacers, inside the jig: one to define each of the two calibration reference points. In this case, the zero-level spacer would be inserted, to set *all* transducers to their respective zero levels, and step 1 would be carried out for each transducer in turn. A similar sequence would then be performed with a different spacer for step 2 and so on.

The calibration program should not, in this case, assume that the whole calibration procedure will be completed for each transducer in turn. The user should be allowed to change sensor channels at any stage between the various calibration steps, in order to begin calibrating another channel. At some later time the user should then be able to resume calibration of the original channel.

Offset correction

As mentioned previously, there are many possible sources of offset, some of which might change over time or with successive repetitions of a measuring process. Offsets can be introduced by factors such as tare weights of containers or other variables which affect the baseline of the measured quantity.

Most measuring systems should be recalibrated periodically. Fortunately, the sensitivity and linearity of many systems remains fairly constant, and in these cases, it may be sufficient to check only for variations in the offset in each channel. This facility is essential in dimensional gauging systems such as that described in the previous section. In these systems a master or reference component is periodically placed in the gauging jig so that the software can measure and subtract out any offsets that might be caused by thermal expansion or sensor movement etc.

If possible, the data-acquisition program should repeatedly check for any drifts that might have occurred in the zero offset of each sensor. This is most easily accomplished in systems which perform repetitive tasks (e.g. component assembly machines on a production line) where the measurand returns to some known starting value after each measuring cycle is completed. This value can be compared on successive cycles in order to detect and correct for any changes in offset.

Generating a precise measurand

Prime calibration requires that the output of the measuring system is determined for a number of precisely known values of the measurand. However, it is sometimes impracticable for the sensor's input to be set *precisely* to any fixed value (e.g. the full-scale limit of the measuring system). In such cases it is clearly undesirable for the calibration software to require any *specific* value of the measurand to be applied, and the operator should be allowed some leeway in selecting or adjusting the calibration reference levels.

The following example may illustrate this point. Suppose that an LVDT displacement transducer is attached to a hydraulic arm. While the operator can accurately *measure* the displacement of the transducer's armature, it might be difficult to *adjust* the position of the hydraulic arm with the degree of precision needed to bring about any specific displacement. If, during a calibration sequence, a reference point must be obtained near to the end of the transducer's range, it would be preferable for the software to allow the operator to set the calibration point anywhere within, perhaps, 90–100 per cent of full scale, rather than demanding that the transducer is set *precisely* to full scale. Provided that the operator enters the value of the calibration point actually used, the software should be able to account for the difference between the ideal and actual values of the measurand when calculating the scaling factor.

Remote indication and control

Interactive calibration is normally straightforward provided that the PC is located close to the measuring system, but if the sensors happen to be positioned in a separate room or high up on the support pillars of a bridge, for example, this procedure can be highly impracticable. The operator may be unable to see any visual display of the sensor's output on the computer's screen. It may also be difficult to continually move between the sensor and PC during the calibration process. However, with a little foresight, the programmer or system designer can circumvent such difficulties with features such as extra large displays, audible indicators, remote keypads or remote numeric indicators or simply by using a portable PC.

Security

It is often important to restrict access to the measuring system's calibration facilities. This can be achieved by means of password protection schemes and file encryption techniques. In some applications

security can be enhanced by appropriate choice of operating system. A discussion of these topics is unfortunately beyond the scope of this book, but Grover (1989) provides a useful overview of cryptography and software security issues in general.

Traceability

It is quite often necessary, for quality assurance purposes, to record precise details of every calibration performed. The identity of the operator who performed the calibration procedure might have to be recorded along with the calibration data itself. It is also usually essential to record which instrument or gauge has been used as the prime calibration reference so that the whole calibration is traceable to a higher level standard. Collet and Hope (1983) discuss the subject of traceability in greater detail.

10 Basic control techniques

Many machines and industrial processes are not inherently self-regulating. These systems usually require some form of control mechanism in order to maintain their operational parameters within predefined limits. A control system serves two purposes. It can preserve some steady operational state or it can be used to facilitate adjustments to the state of the process. Many data-acquisition systems are required to generate control signals in order to, for example, start or stop a process or to implement dynamic regulation.

This chapter introduces some simple software techniques that can be used as a basis for controlling actuators and peripheral devices via the PC. The following material is presented in the context of industrial process control systems, but much of what is said can also be applied to systems for use in laboratory, civil engineering, domestic or other environments.

It is not intended to cover the theory of control systems in any depth. Nor shall we discuss choosing and designing control systems – this is the province of the control engineer. Instead, this section is presented from the point of view of software engineers needing to incorporate control facilities within their data-acquisition programs. The design of control systems is a complex subject which cannot be covered in the space available. Personnel charged with such tasks may need a more detailed understanding of control theory than it is possible to impart here and are advised to consult an appropriate specialist text.

10.1 Terminology

While I have attempted to avoid unnecessary jargon, the use of some process-control terminology inevitably streamlines the text. It is, therefore, helpful to define a few basic terms before proceeding.

Generally, a *process* is some system which we wish to control. It might be a manufacturing process, involving a series of discrete operations such as moving a component into place, lowering a hydraulic ram, applying a quality control marker and then ejecting the component. Alternatively it may consist of some continuous activity such as a chemical reaction. The reaction rate may be dependent upon parameters such as temperature and reactant concentration which have to be accurately and continuously regulated.

Process variables are those quantities that affect the balance of the process and, therefore, its end result. The process will, in general, be characterized by several variables. Some will have a greater effect than others on the outcome of the process, and it is generally these variables that are regulated by the control system. Any process variable that is directly manipulated by a control system is known as a *controlled variable*. There may be other process variables that are not directly controlled. As they can also affect the balance of the process, these uncontrolled variables characterize a *process load* which will affect the way in which the control system maintains the process within desired operating tolerances.

When a process receives some form of control signal, there will be a delay before it responds. This *process lag* may be due to several factors. In a process involving a continuous chemical reaction, for example, the process lag may arise from the thermal inertia of a heated reaction vessel or from the time taken for reactants to flow in or out of the vessel. Similarly, actuators and control mechanisms do not respond instantaneously. Heating elements or mechanical devices such as valves generally take some time to respond to changes in their control signal and they have an associated *controller lag*. Sensing systems also have finite response times (see Chapter 3), and this introduces a *measuring lag*. As will become clear later in this chapter, lag times have a profound effect on the dynamic behaviour of control systems.

10.2 An overview of control systems

Control signals can be issued independently of the current or previous state of the process. This type of control is generally referred to as *open-loop* control as it does not involve any feedback from the process. In *closed-loop* systems, on the other hand, the PC measures one or more process variables and then interprets these measurements in order to decide what control signals should be transmitted back to the process. Any changes in the process brought about by the control signal will then be reflected in subsequent measurements

of the process variables. The whole sample-and-control cycle is repeated in order to maintain the variable(s) within some desired operating range.

A variety of different closed-loop systems are used for controlling industrial processes. These fall into two categories: discontinuous and continuous. Discontinuous controllers respond to changes in the process variable by switching the *control element* (i.e. a device that directly influences the process) from one discrete state to another. A discontinuous temperature controller might respond to a fall in temperature by switching on a heater. When the temperature rises sufficiently, the heating element is then switched off again. Continuous controllers provide a more gradual response and generally are capable of reacting proportionately to both large and small changes in the process variable.

Figure 10.1 illustrates continuous and discontinuous control loops. Both use a PC to convert measurements of the process variable into control signals. The main differences between the two systems arise from the types actuator and PC interface used. The controlling software algorithms will, of course, also differ considerably. These will be discussed in detail in the following pages.

The control element shown in Figure 10.1 is usually an integral part of the process itself. It may be a valve which controls the flow of reactant, or a heating element within a furnace. The actuator, on the other hand, is the mechanism which drives the final control element. It may be an electric motor, solenoid or a hydraulic or pneumatic

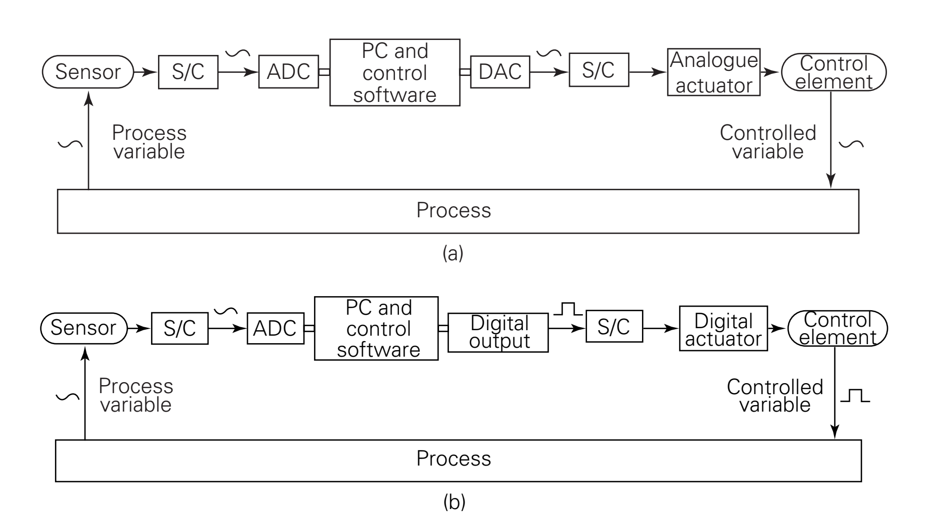


Figure 10.1 *Schematic diagrams of PC-based control loops: (a) continuous control and (b) discontinuous control*

device. It may even consist of a simple relay or electrical circuit which regulates the voltage supplied to the control element (e.g. a heater). Some actuators provide the capability for continuous variation over their working range, while others provide only two-position control.

10.3 Programmable logic controllers

In many cases, the PC drives actuators via analogue or digital outputs and signal-conditioning circuits. In other instances, the process may require a dedicated controller to be used. This may take several forms. Electronic controllers are commonly used to implement continuous analogue control loops. Discontinuous control systems are often built around digital microprocessor-based controllers. These devices, which may be programmed to suit a wide variety of processes, are known as Programmable Logic Controllers (PLCs). Although they are most often used for discrete state process control or machine control, some PLCs have the capability to operate as continuous controllers.

In certain applications, the PC may be required to interface to a PLC. The PLC then directly controls the process while the PC acts in a supervisory role, perhaps monitoring certain process variables, dynamically adjusting set points or logging data for subsequent quality-assurance checks. Communications may be established between the PC and PLC by means of specially designed PLC communications modules which use an RS-232, RS-422 or RS-485 link and a vendor specific communications protocol (such as Allen-Bradley's DataHighway+). Alternatively, status information and commands may be passed between the PC and PLC via relays and suitable digital I/O ports as indicated in Figure 10.2.

Like the PC, PLCs are sequential devices that execute their control programs one instruction at a time. This means that a PLC does not provide an instantaneous response to its inputs. Neither does it respond simultaneously to two or more inputs. The PLC program executes in a continuous loop, scanning its inputs and then evaluating and updating its outputs repeatedly. The loop-execution time varies between different models of PLC and, of course, also depends upon the nature of the control program and the number of I/O channels which have to be processed. Typical loop-execution times are of the order of 2 to 50 ms. Careful programming is required to circumvent problems associated with PLC response times. The system designer must take account of the effect of the PLC's scan time on the control system. He must also be aware of the potential problems which the inherent time lag might introduce when interfacing to the PC.

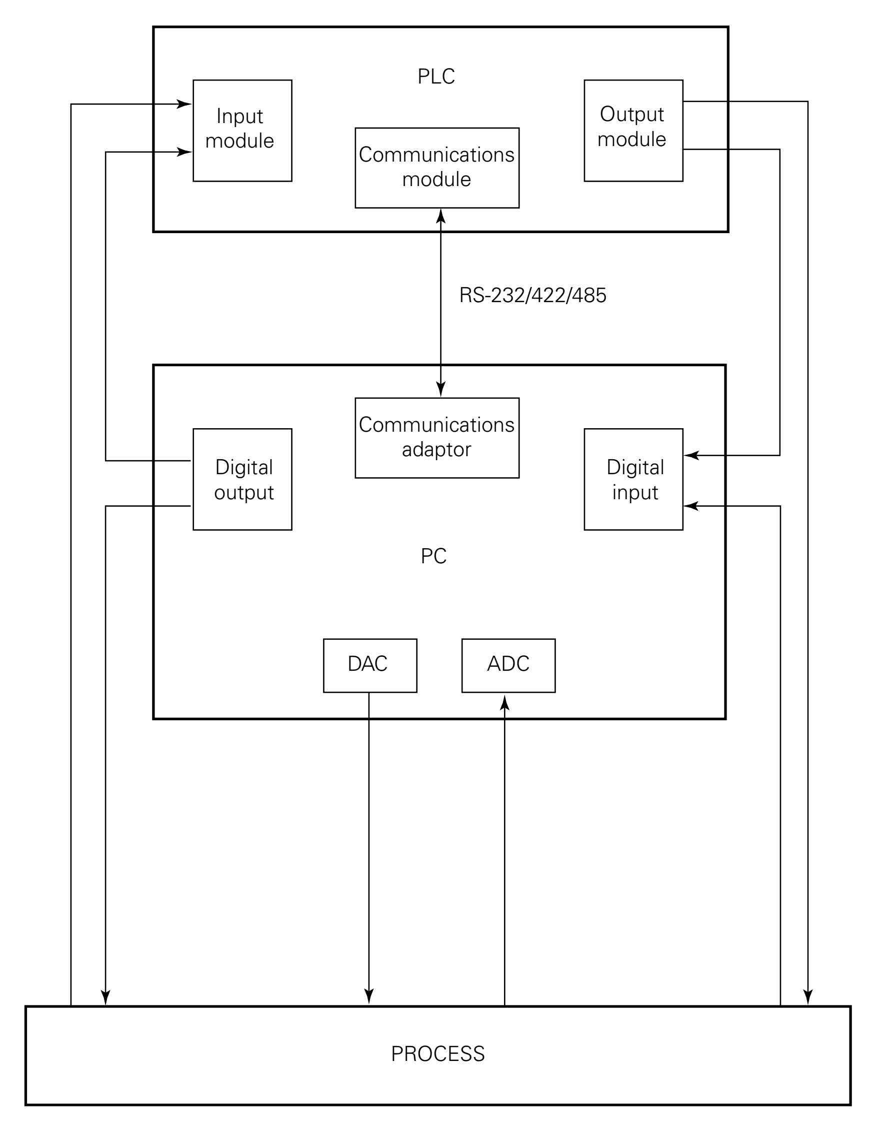


Figure 10.2 *A PLC-based control system using the PC in a supervisory role*

10.4 Safety and reliability of control systems

Before discussing the elements of a control system, we should mention the most important consideration: safety. Many processes are intrinsically hazardous. The consequences of a control-system failure and the ensuing loss of control can sometimes be catastrophic, resulting, at best, in lost production time or at worst in injury or

death. A PC-based control system is founded on a number of complex interacting subsystems which may provide a significant potential for failure. Although various steps can be taken to make the software element of such systems as robust as possible (see Chapter 2), it is still often the most unreliable element. Software-based controllers should not be used in isolation in potentially hazardous or safety-critical applications. Suitable backup mechanisms, processes and quality checking schemes should always be employed to ensure safety in the event of failure of the control system. Determining the types of safety feature appropriate for any given system may require a detailed knowledge of the dynamic behaviour of the process and of the control system itself. This task should be undertaken only by a suitably qualified process engineer.

The following is a list of some basic (and, hopefully, obvious) points which you should bear in mind when programming a PC-based control system.

- Consider the state of the controller's output(s) when power is first applied or when a process is started up. Assess how this will affect the subsequent stability of the system.
- All inputs on which controller calculations are based should be thoroughly range checked in order to prevent invalid data from corrupting the control signal.
- The controller *outputs* may also be range checked, helping to guard against the effects of errors in the control algorithm.
- When testing the system, always attempt to use inputs representative of the actual operational characteristics of the process to be controlled and check the system thoroughly under extreme conditions and with full-scale or out-of-range inputs.
- Be wary of accumulating significant rounding errors from repeated floating-point calculations. This is particularly important when using iterative control algorithms where any calculation errors have the potential to be multiplied many times over. It is prudent to test the software for stability over periods comparable with the expected operating timescale of the system.

10.5 Discontinuous control systems

Because of their simplicity and relatively low cost, discontinuous controllers are popular in a broad range of control applications. As described previously, they operate by simply switching some operational parameter (such as the power supplied to a heating element) between two or more discrete states. Such systems are very amenable to digital control using the PC. A process variable is

monitored via an analogue-input channel interfaced to the PC. This provides a stream of data which is passed to a software comparator (see below) or similar algorithm. The resulting Boolean data is then used to drive the control element via a suitable digital I/O port and actuator.

Most discontinuous control systems operate in a two-position mode offering only two possible states. These states may, for example, determine whether or not power is applied to a heater, or whether a motor is switched on or off etc. Such systems do not respond to variations of the process variable between the two switching levels: they react only when the variable exceeds or falls below one or other of the levels. Other types of discontinuous controller employ three or more discrete switching levels. These might, for example, be used to drive a control element to its zero, halfway or full-scale position.

Software comparators

These are simple routines which compare an analogue value (typically a sensor reading) with one or more predefined *trip levels* (or *set points* as they are sometimes known). The comparator routine generates a Boolean output (i.e. an integer value 0 or 1) depending upon the value of the analogue input in relation to the trip level(s). The comparator's output may then drive a discontinuous control element via a suitable digital output port and actuator.

Comparators generally possess either one or two trip levels. Facilities are often incorporated in the software to allow the trip levels to be adjusted by the end user. Single-trip comparators are suitable for virtually any application where only an upper or lower limit need be applied. They are widely used in discontinuous control systems. They are also often used for starting or stopping a data-acquisition run when data exceeds some predefined level. In addition, they may be applied to elapsed-time readings in order to trigger certain operations or events at appropriate times. In the case of comparators that have a pair of trip levels, the Boolean output might, for example, be set to a 0 when the analogue value falls between the levels, and to 1 when it falls outside. These are used principally for applying tolerance bands to sensor readings (e.g. in pass/fail testing).

Hysteresis and stability

When an analogue signal is close to one of the trip levels, small variations in the signal (e.g. noise) may cause a series of rapid changes in the comparator's output. This is often problematic. If the comparator is used to drive a discontinuous control system it will cause the actuator and control element to repeatedly cycle between

their 'on' and 'off' conditions. Depending upon the nature of the control system, such cycling can result in poor control or excessive wear of the actuator or control element. Another example where noise may cause problems is in real-time displays. If a comparator is used to control the colour of an on-screen digital display or annunciator, any rapid changes in the comparator's output will cause the display to flicker or appear unstable.

These problems can be easily circumvented by introducing a degree of hysteresis into the comparator. Hysteresis is a lag between a change in one variable and some consequent change in another variable – i.e. a lag between cause and effect. It influences the behaviour of a system such that changes occurring in one variable are affected by a 'memory' of the previous state of the system. Hysteresis often manifests itself in real devices or processes by preventing a state change induced by a certain sequence of conditions from being reversed by simply applying the opposite sequence of conditions.

We can incorporate hysteresis into software comparator routines as follows. A neutral zone (or dead band) is applied to each trip level in such a way that the comparator's output does not change state while the input to the comparator is within the dead band. This is illustrated in Figure 10.3 and may be implemented in software as shown in the following code fragment (which is meant to be executed repeatedly within a loop).

```
Output = PreviousOutput;
if (Input > TripLevel + Deadband) Output = 1;
if (Input < TripLevel - Deadband) Output = 0;
PreviousOutput = Output;
```

Note that hysteresis results in the loss of some sensitivity. The technique must be applied with care if the accuracy of the comparator or control system is not to be adversely affected. It is obviously important, when selecting the width of the dead band(s), to achieve a sensible compromise between stability and responsiveness.

It should be remembered that, although it can enhance stability, hysteresis cannot guarantee a smooth controller action. By their very nature, discontinuous controllers affect the controlled variable in a series of discrete steps. Consider, for example, a two-position controller used to regulate the temperature of a furnace. When the temperature rises above some upper limit (equal to the desired temperature plus dead band), the controller switches off the heating element. The temperature then begins to fall until it reaches a predefined lower limit (desired temperature minus the dead band), at which point the controller switches the heater on again. The

procedure is repeated indefinitely, thereby allowing the temperature to cycle between upper and lower operational limits as shown in Figure 10.4.

The presence of any lags in the system mean that an instantaneous response is generally not possible. This will result in cyclic variations of the controlled variable that will exceed the controller's switching levels. One can often compensate for such a behaviour by simply reducing the width of the dead band, although this will tend to make the system more susceptible to noise.

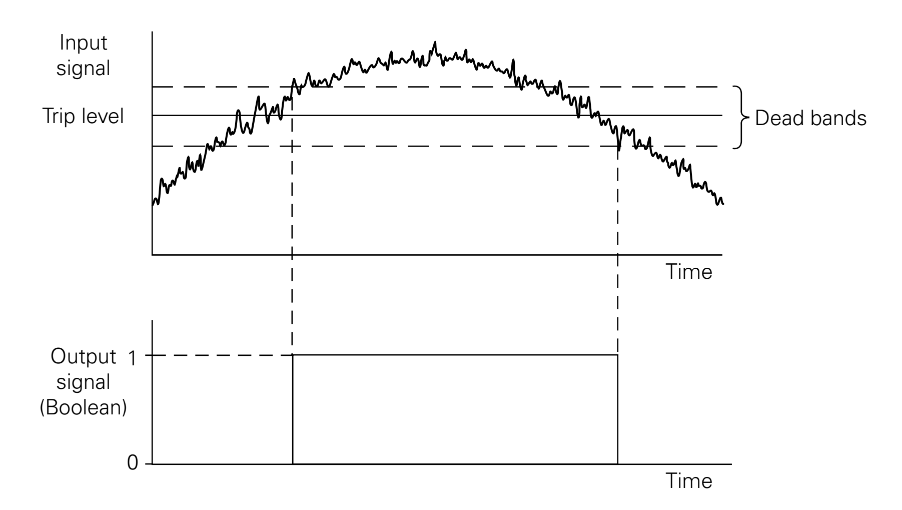


Figure 10.3 *Implementing hysteresis in comparators by means of dead bands*

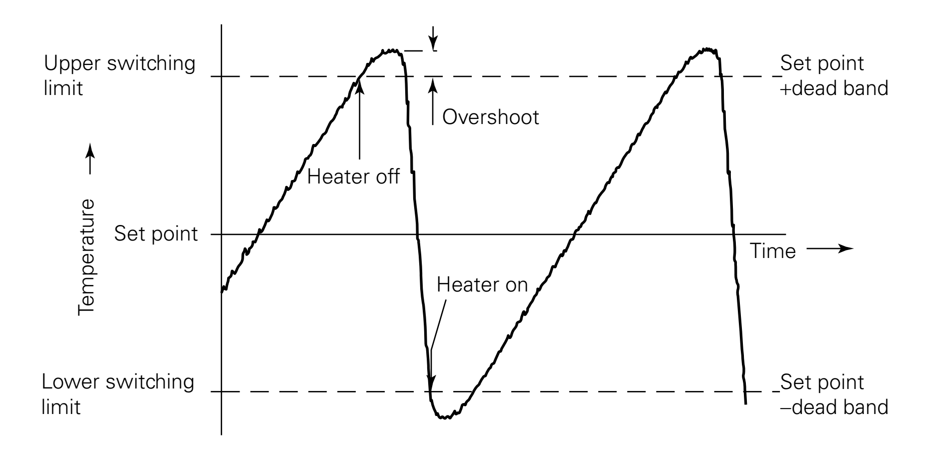


Figure 10.4 *Temperature cycling induced by a two-position discontinuous control system*

10.6 Continuous control systems

Continuous control systems maintain a process variable at or near some desired value by providing a smooth, rather than stepwise, change in the control signal. The process is monitored via an appropriate form of sensor, and a series of digitized and scaled sensor readings is then passed to a continuous-control routine within the software. The output from this routine is scaled, and used to regulate some aspect of the process via a DAC, actuator and control element as indicated previously in Figure 10.1.

A key feature of properly tuned continuous control loops is their ability to provide a timely and proportionate response to process load changes and to transient disturbances. Very small, or slowly changing, deviations can be corrected by a correspondingly small change in the control signal. Many (but not all) continuous control loops are also characterized by the absence of any oscillation in the process variable.

The central question which we must address is: what form does the continuous control signal take and how should it react to changes in the measured variable? There are a number of general-purpose continuous control techniques. Most of these are based on the concept of the error, E, in the process variable. This is the measured deviation of the variable from its desired operating value (i.e. the set point) and is usually expressed as a fraction of the range of allowable input values:

$$E = \frac{v - v_{sp}}{v_{\max} - v_{\min}} \tag{10.1}$$

In this equation, v represents the value of the controlled process variable, v_{sp} is the set point (i.e. the desired ideal value of v) and $v_{\min}$ and $v_{\max}$ represent the limits of the full-scale range of the variable. The error, E, may take either positive or negative values. The signal generated by the control unit (i.e. the PC) is related to the current value of E and/or the history of E values.

Proportional–integral–derivative (PID) control

Continuous control systems generate signals which are some continuous function of E. Often this function is a simple proportionality (i.e. $\propto E$) or is proportional to the integral or derivative of E with respect to time. Proportional, integral and derivative control modes each have specific advantages and disadvantages. Combinations of these three terms are normally used in real control applications.

This not only provides the cumulative benefits offered by each term, it also helps to negate some of the drawbacks of using certain modes (i.e. terms) in isolation. The most generally useful (and widely used) type of continuous control system employs all three modes. This PID, or three-term, controller can be easily modelled in software to allow the PC to manage a variety of process-control applications. The following equation illustrates how a PID controller is formulated.

$$y = PE + PI \int_0^t E\,\mathrm{d}t + PD\frac{\mathrm{d}E}{\mathrm{d}t} \tag{10.2}$$

Here, y is the controller output and is dimensionless; t is the elapsed time, and P, I and D are constants which are chosen to match the characteristics of the control loop to those of the process being controlled. P is known as the proportional gain and is also dimensionless. It controls the scaling of all three terms in the equation. Its sign determines whether the controller provides a direct or reverse action (i.e. whether y increases or decreases in response to an increasing error, E). The contribution supplied by the integral term is determined by the magnitude of the reset rate constant, I. The dimensions of I are time^{-1}. This constant is sometimes expressed in terms of its inverse, known as the reset time or integral time, $T(=I^{-1})$. Similarly, the derivative time constant, D, governs the effect of the derivative term. D has dimensions of time.

Note that, while E remains zero, the contribution from each of the three terms, and hence the controller's output, is also zero. In a practical application this operating point may have to be offset (by adding an appropriate constant) and the controller's output scaled in order to correctly drive the actuator and control element via a DAC. The offset and scaling factors used will be specific to individual processes and control-loop implementations, and will be disregarded in the following discussion.

Programming a PID algorithm

The integral and differential terms in Equation 10.2 can be approximated by the following equation, which may be used with a series of discrete samples. In this equation the n subscript represents the latest sample or calculation, while the i subscript is used simply as an index over which to sum values from all previous iterations of the control algorithm.

$$y_n = PE_n + \frac{PI}{2}\sum_{i=1}^{i=n}(E_i + E_{i-1})(t_i - t_{i-1}) + PD\frac{E_n - E_{n-1}}{t_n - t_{n-1}} \tag{10.3}$$

Here, a simple linear approximation has been used to estimate the derivative term. The integral term is evaluated using the well-known trapeziodal rule. These approximations should be adequate for most control applications, provided that the error is sampled at a rate of more than about ten times the maximum frequency of the input signal. You should, however, assess the accuracy of such an approximation and its potential consequences in your own particular application. If in doubt, it is generally wise to use a sampling rate as high as reasonably achievable, which will help to minimize any errors inherent in the approximation. Indeed, it is a requirement of digital control systems in general (even those without an integral term) that any lags introduced by the controller (in our case, the software) should be as small as possible and this in practice means at least an order of magnitude less than the process lag.

You should also bear in mind the potential effects of timer accuracy and granularity on integral and derivative calculations. Fortunately, most process-control applications require sampling to be carried out at quite low frequencies – often once every few seconds or even every few minutes. In most cases, this rate can be easily accommodated on the PC without incurring any serious problems associated with timing inaccuracies.

Listing 10.1 illustrates how Equation 10.3 may be implemented. The PID calculation is performed by repeatedly calling the `CalcPID()` function and passing a new sample of the process variable, `V`, together with the time, `T`, at which the sample was taken. The time values may be derived from the PC's system clock, RTC or any other convenient source. The controller output, `Y`, is then calculated and passed back to the caller. Each `V` value should be obtained from an appropriate sensor and suitably scaled and/or linearized before being passed to the `CalcPID()` function. Similarly, scaling of the controller output, `Y`,

Listing 10.1 *A simple PID algorithm*

```
/* PID Variables - The following must be initialized before starting PID */
unsigned int FirstLoop;        /* Flag for first loop */
double P;                      /* Proportional gain constant */
double I;                      /* Integral (reset rate) constant */
double D;                      /* Derivative time constant */
double VSP;                    /* Set point */
double VMax;                   /* Maximum input */
double VMin;                   /* Minimum input */
double YMax;                   /* Maximum output */
double YMin;                   /* Minimum output */

/* PID Variables - The following need not be initialized */
double Integral;               /* Summation for integral term */
double LastE;                  /* Last error value */
double LastT;                  /* Last time value */
```

Listing 10.1 *(continued)*

```
void
CalcPID(double V, double T, double *Y)
/* This function calculates the PID controller output, Y, for the new value of
   the variable V at time T. The first time that this function is called it
   returns Y = 0.
*/
{
double E;
double DeltaT;

/* Check V is within its specified limits */
if (V > VMax) V = VMax;
if (V < VMin) V = VMin;

/* Calculate the error, E */
E = (V - VSP) / (VMax - VMin);

if (FirstLoop)
       {
       Integral = 0;
       *Y = 0.0;
       FirstLoop = 0;
       }
  else {
       DeltaT = T - LastT;
       Integral = Integral + DeltaT * (E + LastE);
       *Y = P * (E + I * Integral / 2.0 + D * (E - LastE) / DeltaT);
       }

/* Clip controller output to required range */
if (*Y > YMax) *Y = YMax;
if (*Y < YMin) *Y = YMin;

/* Update record of last E and T values */
LastE = E;
LastT = T;
}
```

will also be necessary before outputting the signal via a DAC to the actuator and control element.

A number of global variables are declared at the beginning of the listing. Several of these must be initialized before calling the `CalcPID()` function for the first time. The `P`, `I` and `D` variables are simply the PID constants defined previously. `VSP` is the set point for the process variable. `VMax` and `VMin` specify the range of the process variable (they are the same as v_{max} and v_{min} in Equation 10.1), while `YMax` and `YMin` define the limits of the controller's output range. The `FirstLoop` variable should also be initialized to 1 before commencing a sequence of PID calculations. This variable acts as a flag to prevent the `CalcPID()` function from attempting to perform a PID calculation the first time that it is called. The `Integral`, `LastE` and `LastT` variables

are initialized automatically by `CalcPID()` and may be left undefined by the caller.

You should ensure that the units of all variables are consistent. The derivative time, D, and the sample time, T, have dimensions of time and should be allocated units of minutes or seconds. The reset rate constant, I, should be expressed in minutes^{-1} or seconds^{-1} as appropriate. Listing 10.1 can be modified in practical applications to incorporate the global variables as fields within a structure or object. By allocating a separate instance of the structure (or object) to each process variable, the same function may be used to operate several PID loops.

The technique employed for the PID calculation will accommodate slight variations in the sampling rate (provided that those variations are accurately reflected in the `T` values passed to the `CalcPID()` function). It also employs the trapezoidal method of calculating the integral term. However, if we fix the sampling rate and employ a *rectangular*, rather than trapezoidal, approximation for the integral (i.e. each discrete panel in the $E(t)$ function is approximated by a series of rectangles of height $E(t_i)$) it is possible to greatly simplify calculation of the controller output. In this case, the nth output is given by:

$$y_n = P \quad E_n + I \sum_{i=1}^{i=n} E_i \Delta t + D \left(\frac{E_n - E_{n-1}}{t_n - t_{n-1}} \right. \tag{10.4}$$

where Δt is the time interval between successive samples. An equation of the same form can also be written for the controller output obtained at the previous stage, y_{n-1}. Then by subtracting the expression for y_{n-1} from that for y_n we obtain:

$$\begin{aligned} y_n = y_{n-1} + E_n \quad & P + PI\Delta t + \frac{PD}{\Delta t} \\ - E_{n-1} \quad & P + \frac{2D}{\Delta t} \quad + E_{n-2} \quad \frac{PD}{\Delta t} \end{aligned} \tag{10.5}$$

The terms in brackets consist simply of constants and can be evaluated before commencing the PID calculations. This formula is often used in computer-based PID controllers as the basis of an iterative control method. It is somewhat simpler than Equation 10.3 and requires calculation of only the *change* in controller output at each step. It is a simple matter to adapt Listing 10.1 for use with Equation 10.5.

There is an obvious, although sometimes overlooked, consideration when designing a PC-based continuous control system. The

discrete nature of the digitization processes inherent in reading samples and outputting control signals may limit the accuracy of the control loop. This can prevent the system from achieving a steady equilibrium ($E = 0$) state and can cause the controlled variable to fluctuate by an amount equivalent to the combined resolution of the measuring and control subsystems.

Characteristics of the P, I and D terms

It is instructive to briefly examine the contribution that each of the three PID terms makes to the controller's output. The proportional term (also known as the modulating term) provides a smooth linear response to changes in E. As shown in Figure 10.5, the proportional response curve saturates at the extremities of the controller's output range. Thus, there is a limited range of E values – known as the proportional band – over which proportional control is maintained. In the absence of any contribution from the other terms, an error value of $E = 0$ is usually chosen to generate an output halfway along the controller's range.

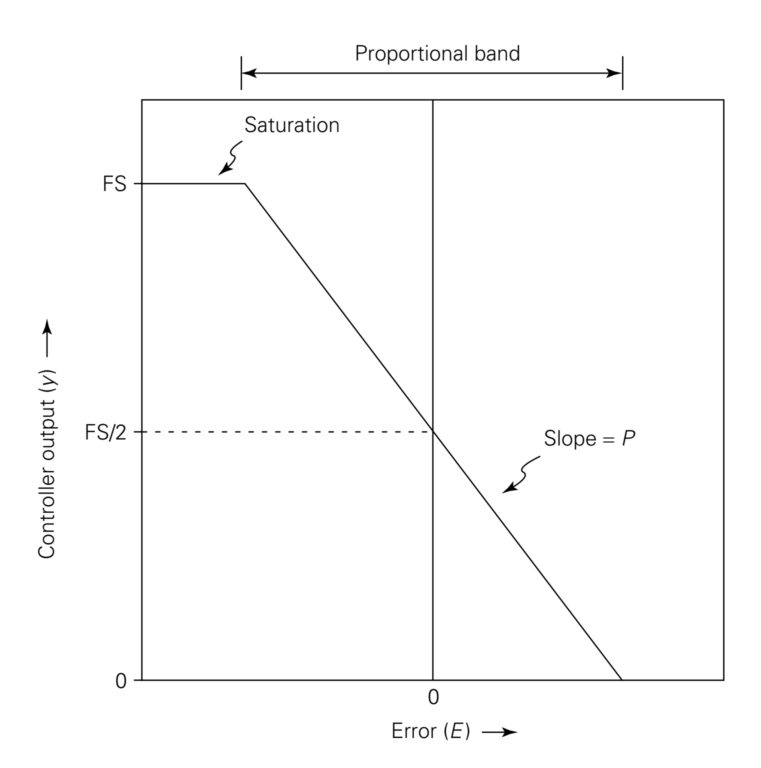


Figure 10.5 *Contribution from the proportional term*

The slope of the curve is simply the proportional gain constant, P, and may be either positive or negative (a negative slope is shown). Larger values of P will lead to a smaller proportional band. They can also give rise to oscillations in the controlled variable. If P is too large, it can cause the controller's output to overshoot the desired setting, which may result in cycling of the process variable. An element of proportional control is, however, usually desirable as it gives a one-to-one correspondence between the error and controller output and has an effect which is independent of the frequency at which the error changes.

The contribution from the integral term *changes* at a rate proportional to E. It increases (or decreases) steadily during periods when E is non-zero. Positive errors cause the output to increase while negative errors cause it to decrease. The longer that E deviates from zero, the greater will be the controller output. The inverse of the reset rate constant, I, is actually the time taken for the integral contribution to duplicate the proportional output. The integral term is capable of providing a response to large errors and has a greater effect on low frequency variations in E. It is extremely useful in control systems that are subject to sizeable load changes. Without this term, a large proportional gain would be required in order to maintain E within some desired range, and this may induce cycling of the controlled variable.

While the integral term provides a slow response to long-term trends, the derivative term responds quickly to transient disturbances in the controlled variable. It supplies an initial response to sudden changes in E which, if left unchecked, might quickly give rise to larger deviations from the set point. For this reason the derivative mode is also sometimes referred to as *anticipatory control*. An important property of the derivative term is that it provides a degree of damping. This helps to suppress oscillations that tend to occur when a high proportional gain is used in systems with large process lags.

The derivative term cannot be used alone, because it always provides a zero output when E remains constant. It does not reflect the magnitude of E. A large but constant error would still give rise to a zero derivative term. The derivative term will also accentuate any noise present on the input signal, so steps should be taken to minimize noise amplitude. Care should be taken when filtering the input signal to ensure it does not excessively suppress any real high frequency variations in the process variable.

The contributions made by each term, in response to a load change and change in E, are illustrated in Figure 10.6.

The proportional mode exhibits one characteristic which precludes its use, in isolation, in some PID systems. If the process load

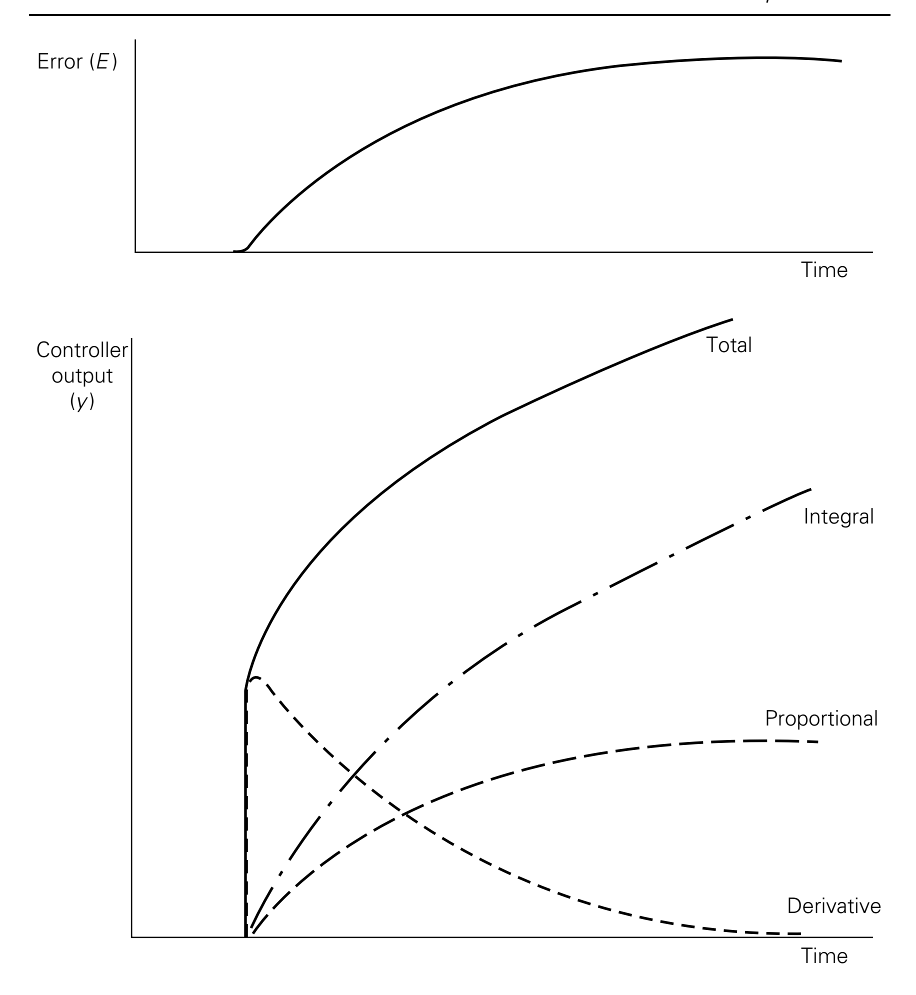


Figure 10.6 *Contributions of the P, I and D terms*

changes, this will induce a non-zero error, E. The controller's output will then automatically adjust to maintain a zero error. As we are dealing with a semi-permanent load change, rather than a transient disturbance, the level of controller output required to achieve zero error will then be offset from its nominal (halfway) point towards one end of its operating range, thereby asymmetrically truncating the proportional controller's operating range. The integral term helps to eliminate the effects of this proportional offset. If a load change occurs that would require a shift in controller output to maintain E at zero, this shift can be provided (after a certain integration time) by the integral term. This consideration is of most importance in PID control systems implemented using separate electromechanical or

pneumatic controllers. It is of less concern in computer-based PID systems as the three terms are not individually constrained within limited operating ranges.

The proportional and integral terms can be used in isolation under certain circumstances. More useful, however, are the combined proportional–integral (PI) and proportional–derivative (PD) modes. The former tends to be suited to systems with large, but slow, changes in process load. The PD mode is capable of dealing with rapid changes in load. Equations 10.3 and 10.5, and Listing 10.1, may be adapted for PI or PD control by setting the coefficient of the unwanted term to zero.

Tuning PID loops: a brief overview

In order to provide a stable and responsive control mechanism, control-loop characteristics must be matched to the dynamic behaviour of the process. This requires PID loops to be properly tuned by careful selection of parameters such as the sampling rate and the P, I and D constants. Designing, tuning and maintaining a control loop can be a complex task, requiring a detailed knowledge both of the specific process and of control-loop optimization techniques in general. This section does not attempt to describe tuning techniques in any depth. The texts by Edgar (1996), Johnson (1988) and Wightman (1972) provide a good introduction to these and related topics. The intention here is to present an overview of the operations and activities involved in control-loop optimization and thus to enable the DA&C programmer to comprehend any related facilities that may need to be incorporated in control software.

The transfer function

Any signal applied to the input of a control system will be modified in some way before being fed back to the process. In general, this modification incorporates two components: amplification and phase shift. We can define a transfer function that embodies the frequency dependence of both of these components. The transfer function of a process and associated control loop is, in many cases, not amenable to analytical representation. Empirical techniques must be employed to assess the behaviour of the control loop.

The transfer function must be such that the controlled variable remains stable at all frequencies. Instabilities arise if the gain and phase shift of the transfer function at any one frequency are such that the feedback signal from the controller tends to reinforce a periodic disturbance. If this were to happen, the magnitude of the disturbance (and, therefore, of the error E) would increase

unchecked. The parameters of the control system must be chosen so as to prevent any such instabilities. They must also be chosen to provide the best possible degree of control. The criteria used for determining the optimum control conditions will vary to some extent between different applications.

Response to a load change

Whenever a load change occurs, the control system should act to restore the process variable to its set point. A stable, controlled process variable may exhibit one of several types of behaviour in response to a change in process load. The type of response depends upon the parameters of the control loop and upon the process and controller lag times.

The response of the controlled variable over time is of interest as it provides a measure of the efficiency or quality of the control loop. Oscillations or cycling of the process variables sometimes occur as shown in Figure 10.7. This phenomenon arises when the system is underdamped and it results in a periodic deviation of the process variable about the set point. Oscillations may also occur when some control systems are started up. An overdamped system on the other hand will not oscillate when subjected to a load change, but it may take an unacceptably long time to restore the variable to its set point.

It should be clear that, whether the system is underdamped, overdamped, or critically balanced between the two regimes, it will

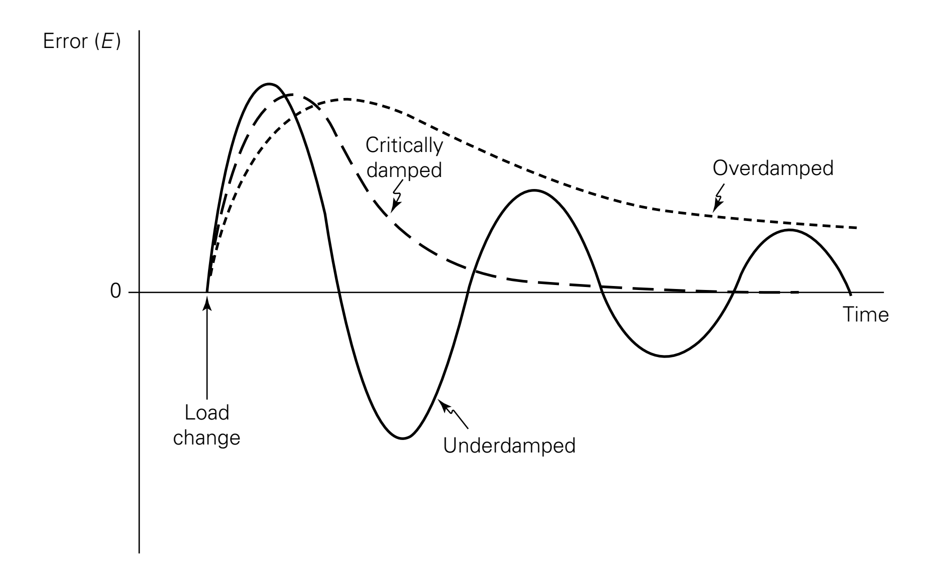


Figure 10.7 *Responses of a stable, controlled variable to a load change*

never be possible to *instantaneously* restore the process variable to its set point. The best that can be done is to ensure that the control loop is tuned, by careful selection of the *P*, *I* and *D* constants, so as to provide the best possible degree of control. This means minimizing the deviations of *E* from zero and also minimizing the time intervals during which *E* falls outside the desired tolerance band.

PID tuning methods

Several methods can be used to determine the optimum values of the PID constants. Some methods involve measuring the phase shift and gain components of the transfer function over a range of frequencies. The transfer function is then repeatedly modified by adjusting the controller's *P*, *I* and *D* constants until the desired functional form is obtained. Other methods, however, involve a more empirical approach in which the transfer function is not explicitly determined.

One such technique, known as the open-loop response method, may be used only in inherently stable and self-regulating processes. This requires the control loop to be broken, by disconnecting the controller's output from the actuator and control element. A small disturbance is then manually induced in the control signal and the process variable should then change in response to the disturbance. The rate at which it changes, the magnitude of the change and the time lags inherent therein are then measured, and the optimum values of *P*, *I* and *D* are calculated from these parameters.

Another technique, which *is* more suited to processes that are not inherently self-regulating, leaves the control loop intact. This method, known as the Process Cycle method, induces oscillations of the process variable about its set point. The cycling characteristics are first measured and then used to calculate the optimum values of *P*, *I* and *D*. The method involves setting the derivative and integral constants to zero and gradually increasing *P*. Small transient disturbances are also regularly applied to the process in order to trigger oscillations. When steady oscillations finally begin, their frequency and the proportional gain at which the oscillations started can be used to calculate the optimum values of *P*, *I* and *D*.

You should refer to a process-control text such as Johnson (1988) or Edgar (1996) for the formulae required for calculating the PID constants. The formulae used in any program for calculating *P*, *I* and *D* should always be specified by a qualified process engineer. It is not appropriate to discuss details of such calculations here – indeed, they might vary somewhat between different applications. Instead we will make a few general comments on the facilities that you might

need to include in your software in order to facilitate control-loop optimization.

Software facilities

The first thing that should be borne in mind is that (as previously stated) many process-control systems are so complex that a simple analytic calculation of the transfer function is not possible. Consequently, the tuning techniques which have to be employed are at least semi-empirical and generally involve a degree of informed trial and error. Second, set-point changes or changes in the PID constants of any one control loop may have an effect on the behaviour of other interacting process variables. The process engineer will usually need to monitor the state of at least one, and possibly several, process variables after any changes to the control system are made.

For both of these reasons, an interactive approach such as that advocated for calibration and linearization in Chapter 9 is likely to be one of the most usable solutions. This might allow the state of the process to be continually monitored on screen, while adjustments to the control parameters are made via the keyboard or mouse. Graphical chart-recorder type displays, showing the history of one, or more, process variables, may be required, together with numeric representations of the current readings. Certain derived quantities may also be of interest. Displays showing the maximum value of E, the lengths of time during which E exceeds acceptable limits, or the total accumulated error (i.e. the integral of E) over user-defined intervals may be needed in order to assess the quality of the control system.

When tuning closed-loop systems, it may also be necessary for the software to incorporate facilities for measuring cycling frequencies, process and controller lags, and phase differences between the process-variable inputs and the resulting controller outputs. These parameters are required in order to determine the optimum values of the PID constants. Other facilities, such as the ability to label points, insert comments into the process-history graphs and to log the process variable data to disk may also be helpful in some instances.

In addition, it is possible for the software to provide a degree of automation in the tuning process. Open-loop tuning techniques are facilitated if the control loop can be broken within the software, thereby removing the need for physical disconnection. Oscillations or periodic disturbances which might have to be applied to the process can, in some cases, be generated via the software. Clearly, the facilities required and the details of their implementation will depend upon the nature and complexity of the process to be controlled.

This Page Intentionally Left Blank

Part 5 Examples

This Page Intentionally Left Blank

11 Example projects

This chapter presents several examples, based on real projects, which illustrate how some of the topics discussed so far can be applied. A few of the examples are parts of much larger systems or suites of data-acquisition programs, and a complete analysis of (and justification for) certain elements of the design cannot be presented in the space available.

Also, bear in mind that the projects described here are merely representative examples of typical applications. Data-acquisition techniques may be applied to a diverse range of measurement tasks, such as fuel-flow monitoring, bridge jacking, strain measurement, control of rolled sheet metal production, pile testing or brick manufacturing. You may encounter many others. It is important to remember that the techniques described here will not always be the most appropriate solution for your own applications.

The examples presented encompass measurement of displacement, load, torque, temperature and light intensity; dynamic sampling issues; linearization; cold-junction compensation; interrupt-based I/O; serial I/O; discontinuous control, and PLC interfacing functions. We will begin with two examples illustrating some of the practical problems associated with sensor calibration.

11.1 Dimensional gauging of railway carriage wheels

An example of how linearization techniques can be applied to overcome deficiencies in the sensor's response and poor measurement geometries.

Overview

The purpose of the project was to provide instrumentation and software for a portable gauging system intended to measure the

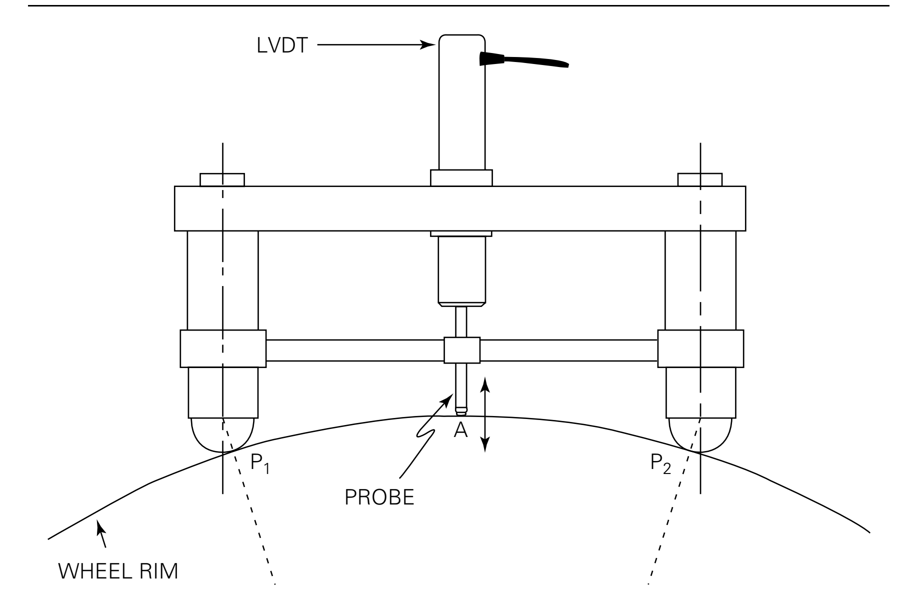


Figure 11.1 *Apparatus for measuring railway carriage wheel diameters*

radius of curvature (and thus the degree of wear) of railway carriage wheels. The client's gauging jig consisted of a rigid frame into which the body of an LVDT displacement transducer was to be fixed (see Figure 11.1). The displacement of the tip of the probe (point A in Figure 11.1) was measured relative to two fixed reference points P_1 and P_2. As all three points were in contact with the wheel rim, the measured displacement provided an indication (albeit a non-linear one) of the wheel's radius of curvature.

The LVDT was coupled via a portable signal-conditioning module and 16-bit ADC (PCMCIA) card to an 80486-based laptop PC. The principal programming task was to derive linearization factors that could be used to convert the probe displacement (ADC counts) to a readout of wheel radius. These factors were readily obtained from an analysis of the geometry of the apparatus.

Special problems and considerations

The client had designed the measurement geometry such that a large range of wheel diameters (0.7–1.8 m) could be encompassed by a relatively small displacement of the LVDT (full-scale range 50 mm). This introduced two problems. First, there was the potential for small pits and irregularities in the vicinity of points A, P_1 and P_2 to significantly affect the accuracy of the system. Second, the very small

inaccuracies inherent in the LVDT and electronic components were expanded into relatively large uncertainties in wheel diameter. The former problem could be circumvented to some extent by training the operators to avoid irregularities on the wheel rim and to average several readings taken at various positions around the rim. The latter consideration was more problematic.

The principal source of inaccuracy in the electronic components was the non-linearity of the LVDT itself. This was minimized by using an LVDT with parallel coil geometry. The smooth response of this type of transducer can, when linearized using a power-series polynomial, offer a greater precision than an LVDT with conventional stepped windings (see Chapter 9). In this case the non-linearity of the LVDT was reduced to 0.05 per cent of full scale, leading to a theoretical precision of ±0.2 mm and ±1.5 mm in the wheel radius readout for diameters of 0.7 m and 1.8 m respectively.

These figures were difficult to realize in practice, however. The effects of thermal expansion on the gauging jig and LVDT can easily introduce significant errors in the gauge's output. Nevertheless, provided reasonable handling precautions and environmental restrictions were observed, the device was able to provide the required degree of accuracy and repeatability.

11.2 *In-situ* sensor calibration on a tube-straightening machine

An illustration of some of the problems encountered during *in-situ* calibration of sensors in a production environment.

Overview

The client required software for *in-situ* calibration of a multi-channel array of displacement transducers used on a tube-straightening machine. The machine possessed 16 sets of angled rollers, through which lengths of steel tube were passed after manufacture. The positions of the rollers were varied hydraulically under computer control in order to remove bends from the tubes. A series of displacement transducers (two per roller set) was used to monitor the position of the rollers.

The displacement transducers were interfaced to an 80486-based PC via an intelligent 64-channel data logger (only 32 channels of which were used). The data logger provided transducer excitation, signal conditioning and a 16-bit ADC. The unscaled ADC readings

were transmitted back to the host PC via an RS-232 link. The PC was in turn interfaced to the tube-straightening machine via three 16-bit digital output cards, an accessory relay panel and PLC. In this way, the roller positions could be varied and monitored by the control program running on the PC.

Because vibrations, temperature fluctuations and other environmental factors could influence the accuracy of the roller position readings, a facility to periodically recalibrate the displacement transducers was required. A separate program was used on the PC for this purpose and it is the design and operation of this element of the system that we shall concentrate on here.

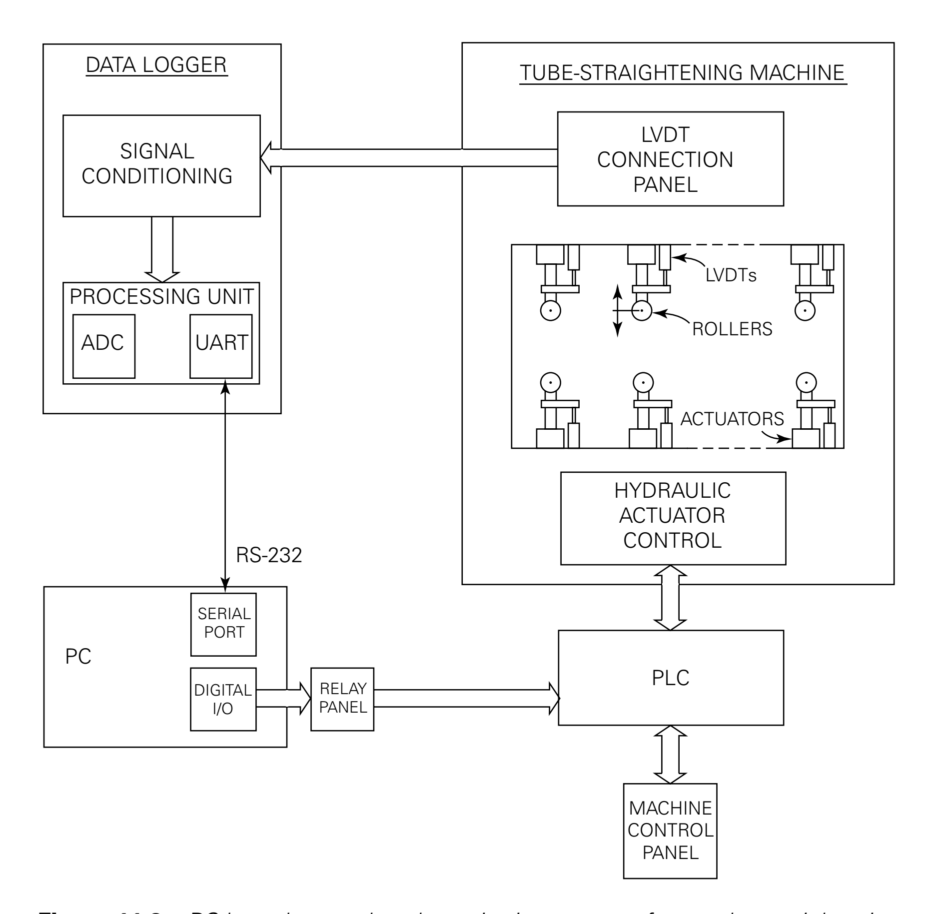


Figure 11.2 *PC-based control and monitoring system for a tube-straightening machine*

Calibration program

The function of this program was to perform two-point linear calibration on each displacement transducer. The calibration factors determined in this way were then stored in a space-delimited ASCII file on the PC's hard disk, and were subsequently used by the machine control program to convert ADC readings to values of roller position in millimetres.

The program was designed to run under Microsoft MS-DOS version 5 and was written using a combination of Borland Pascal and assembly language, the latter being used to implement a serial port driver.

The interface to the data logger consisted of a half-duplex RS-232 link running at 19 200 baud. The serial port driver programmed the PC's UART directly (i.e. at the register level) in much the same way as the example given in Chapter 8. This is a particularly simple task in real mode, which is fortunate because it is not possible to obtain 19 200 baud via DOS's own serial port services. The serial port driver employed interrupt-driven reception techniques. Transmission, being less time critical in this instance, was initiated when required from the main program thread, rather then from an interrupt handler. A 10 ms delay was inserted after transmission of each character as a simple means of avoiding overrun errors in the data logger's UART.

The communications protocol employed by the data logger consisted of a proprietary high level ASCII command set incorporating 16-bit true-binary data transmission. Flow control was implemented entirely in software using a combination of simple timing techniques and echoing of a special acknowledgement character.

In-situ calibration was potentially time consuming as it required each of the 32 displacement transducers to be set manually (via the machine's control panel) to both limits of its range, and for these displacements to be independently measured by some mechanical means (gauge blocks or dial gauges). The measured displacements were then entered into the PC, at which point they could be compared with the ADC readings in order to calculate the calibration scaling factors.

Frequent recalibration of the displacement transducers was deemed to be necessary in the initial stages of development and operation, i.e. until the long-term stability of the equipment could be proved in a production environment.

To minimize lost production time, it was important for the calibration program to be as easy to use as possible and to reduce the

likelihood of operator errors. To this end, an interactive approach was adopted. The program was designed to lead the operator through the calibration process, providing prompts to indicate the next operation required of the user. Commands were entered via a simple menu, which allowed calibration data to be reset and new calibration reference points to be sampled. All numeric data entered by the operator were range checked and facilities were provided for the operator to edit or re-enter the data. A number of other features were incorporated into the program to simplify the calibration procedure:

- Large digital readouts and bar graphs were displayed on screen to indicate the current ADC reading and, when appropriate, the corresponding scaled displacement reading.
- The operator was allowed a degree of latitude in selecting the displacement values that would be used as calibration reference points. This simplified the adjustment of the transducer/roller assembly, which was controlled via a powerful hydraulic system and could be varied only in coarse steps.
- The operators found it simpler to reposition all of the rollers in one operation. To accommodate this, the software allowed the lower calibration reference points to be sampled for all transducers before requiring the upper calibration reference points to be obtained, rather than requiring both the upper and lower reference points to be obtained for each transducer in turn. The distinction between the two sampling sequences was a minor one in terms of software structure, but it had a profound effect on usability.

11.3 Dimensional gauging of turbine blades

This is a particularly interesting example of a technique that is widely used for checking the dimensions of castings or other components with complex shapes.

Background

Because of the very high speed of rotation inherent in aircraft engines, the geometry of the engines' turbine blades is critical. In order to avoid turbulence in the air flowing across the blade's surface its dimensions and shape have to be controlled very precisely during manufacture. Verifying the dimensions of each blade is quite an involved task because of its complex shape. The thickness of the aerofoil portion of the blade varies along its length and width; the upper and lower surfaces are both precisely curved, and the blade is twisted along its length.

Overview

To measure the dimensions of the blade, it is placed in a gauging jig where its aerofoil portion rests on the tips of three datum probes. A number of gauging probes are then brought into contact with the upper and lower surfaces of the blade and their displacement (relative to the plane defined by the three datum probes) is recorded. The probes are positioned in a grid-like structure across the blade's surface and are arranged in pairs (one on the upper surface and one on the lower) so as to facilitate measurement of the blade thickness (see Figure 11.3).

In this instance, the probes were high precision gauging LVDTs, each possessing a full-scale range of 2.0 mm. These were connected to a 256-channel data logger, which provided excitation, signal conditioning and 16-bit digitization. As in the previous example the data logger was interfaced to the host PC via a half duplex RS-232 link and the same ASCII communications protocol was employed. A baud rate of 9600 was chosen in order to accommodate slightly longer RS-232 cables.

Each displacement reading was required to be accurate to within ±0.02 mm. As is often the case in this type of application, the main contribution to the total inaccuracy of the system arose from the linearity of the gauging transducers (0.004 mm). The signal-conditioning and digitization modules of the data logger contributed comparatively small inaccuracies. As this was a static data-acquisition system (i.e. the parameters being measured do not vary during

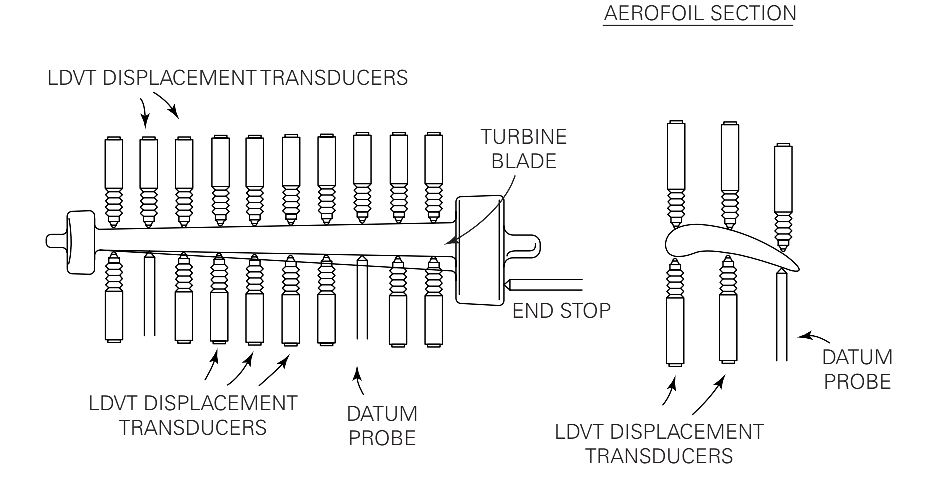


Figure 11.3 *Dimensional gauging of turbine blades*

the time required to sample them), the dynamic behaviour of the system (aperture time, aperture error etc.) was not an important consideration.

The data-acquisition software was designed to run on a 350 MHz Pentium II PC under Microsoft Windows NT 4. It was written in ANSI C using National Instruments' LabWindows/CVI version 5 development environment. This allowed the serial communication routines to be implemented easily using the RS-232 library supplied with LabWindows/CVI. The facilities offered by the Windows GUI were also highly beneficial in this instance because of the high proportion of user I/O required.

Gauging procedure

In the late stages of manufacture, the surfaces of the turbine blades are repeatedly etched, ground and buffed until they attain the desired shape. After each etching or buffing step, the operator places the blade in the gauging jig and the PC records the displacement of the upper and lower surface of the blade at each probe position. Depending upon the stage of manufacture, other parameters such as thickness, twist angle and rate of twist are calculated and compared against predefined tolerances. The probe position and derived data are then displayed on a graphical representation of the blade, and out-of-tolerance readings are highlighted. Using this information, the operator is able to adjust the amount of etching or buffing applied in the next stage.

The display also includes a number of other features such as pass/fail indicators, the maximum and minimum thickness still to be removed from the blade and recommended strength of etchant to be used.

One of the most important benefits that automation of the gauging procedure affords is the ability for the PC to maintain a record of the current stage of manufacture of each blade and to store detailed size and shape information. This data is extremely useful for quality control purposes and is collated and analysed by the client using a commercial SPC (Statistical Process Control) software package.

Configuring the system

The gauging software and the multi-channel data logger were both designed to be highly configurable. By removing or adding gauging LVDTs and signal-conditioning modules (and also replacing the gauging jig), different models of turbine blade could be accommodated. The software incorporated facilities for setting up the system

for use with different blade geometries and probe configurations. A number of parameters had to be defined prior to gauging. These included:

- blade model number
- blade orientation
- datum surface (upper, lower, convex or concave)
- probe grid size and distribution
- probe channel assignment and calibration
- probe orientation (parallel orientation or normal to the nominal blade surface)
- datum probe positions
- display options (i.e. whether certain types of display will be shown during gauging) and
- tolerances for each probe position and thickness reading.

Probe calibration and zeroing

During high precision dimensional gauging, it is particularly important that the probes and their mountings are mechanically stable. Although errors due to thermal expansion or movement of the probes in their mountings are small, they are not always insignificant. One must also remember that the output from gauging transducers and other electronic components are liable to drift slightly over time, particularly in response to temperature changes. The software was designed to accommodate these variations by allowing the probes to be periodically recalibrated and rezeroed.

Initially, all probes were calibrated against a precise standard before being mounted into the gauging jig. A three-point prime calibration technique was used (see Chapter 9) and the PC recorded the scaling factor and zero offset for subsequent use during gauging. A reference blade (with accurately known dimensions) was then placed into the gauging jig and the probe-offset readings were displayed on screen. Each of the probe mountings was then adjusted so as to give an offset reading of zero.

Fortunately, the probe (LVDT) scaling factors tend to be relatively stable, so in this case the full calibration procedure had to be carried out only infrequently. A much greater potential source of measurement error affects the probes' zero positions. This arises due to thermal expansion of the mechanical components and movement of the transducers in their mountings. For this reason, the software enforced a strict zero-offset checking regime in which the operator was required to periodically verify the accuracy of the system against a reference blade.

The system supervisor would enter a limit on the number of gauging operations that could be performed (typically 20) before the operator would be forced to check the probe offsets. If any offset exceeded a predefined limit, the operator would be warned and asked to confirm acceptance, in which case the software would record the probe offsets and use them to correct all subsequent displacement readings. If any of the offset readings was found to be greater than a second predefined limit (indicating that a probe had moved appreciably in its mounting) further gauging would be prohibited until the fault had been rectified.

11.4 Torsional rigidity testing of car bodies

This is another, rather specialized, example of a multi-channel static data-acquisition system. Although the nature of the application is rather different, this project is based on a similar configuration of data-acquisition hardware to that used in turbine blade gauging.

Background

One of the numerous tests required during development of a new model of automobile is to determine the torsional rigidity of its body shell. This is effectively the resistance to a twisting moment applied between the axes of the front and rear wheels. The client had, for many years, been performing these tests manually. The body shell was clamped to a test rig that could be adjusted hydraulically to apply various torques between the front and rear wheel axes. Up to 80 dial gauges (devices with analogue dial readouts, used for measuring linear displacements) were distributed symmetrically about the centre line of the body shell. The applied torque was increased in a number of steps and at each stage the displacements registered by the dial gauges were recorded manually (using pen and paper). Readings were then taken over a decreasing range of torque values until the torque returned to its initial value of zero and any residual deformation would be recorded. In some cases the whole cycle of measurements would be repeated several times using both clockwise and anticlockwise twisting moments.

Overview

Because the measurement process was carried out manually, it was very time consuming and potentially error prone. The client wanted to automate the data-gathering procedure by substituting linear

displacement transducers for the dial gauges and in this way to record the displacements and applied load electronically.

The client wished to introduce the electronic components gradually, both for reasons of cost and to allow the performance of the new technology to be verified against the old measuring system. Initially, only about half of the dial gauges were to be replaced by electronic sensors. The remainder would still be read manually, but the data would now be entered directly into a handheld electronic keypad. The most appropriate solution (at the time of developing the system) was a Psion Organiser II. This was a small, programmable, battery powered unit with an alphanumeric keypad. It had sufficient memory to store all of the data required as well as a specially written data-entry program, and possessed an RS-232 interface for downloading data to the PC.

The PC itself was equipped with a 25 MHz 80386 processor, 4 MB of RAM, 80 MB hard drive, one Centronics parallel port and two RS-232 serial ports. One serial port was used for the Psion Organiser Comms Link interface, the other for linking to an intelligent multi-channel data logger.

Data-acquisition hardware

The data logger was equipped with signal-conditioning and excitation modules for up to 80 LVDT and eight strain-gauge-bridge transducers. Only one of the strain-gauge-bridge channels was used and this was connected to a tension/compression load cell with a full-scale measurement range of ±2500 N. This channel was scaled, in accordance with the geometry of the torsion rig, to generate torque readings of ±5000 Nm, accurate to ±25 Nm. LVDTs with various full-scale ranges were used for the displacement measurements. Each possessed a linearity figure better than 0.25 per cent. All other sources of inaccuracy in the electronic components of the system were comparatively small and could be ignored. As in the previous two examples, the data logger communicated with the PC via a half-duplex link using a proprietary ASCII communications protocol.

Data-acquisition software

The software was designed to run under Microsoft MS-DOS version 5 and had three principal components: calibration routines, test configuration facilities and the data-acquisition routines.

All of the displacement sensors were removed from the test rig prior to calibration. The LVDT displacement transducers were calibrated against a precision micrometer standard, using an interactive linear three-point technique (see Chapter 9). Three-point prime calibration is a particularly appropriate method because LVDTs possess an intrinsic null position at the centre of their measurement range (see Chapter 3). Prime calibration techniques were also employed for load cell calibration. The scaling factors and zero offsets for each channel were recorded in a binary file on the PC's hard disk for use during subsequent torsion tests.

The test configuration routines allowed the operator to specify all of the parameters needed to identify and automate each test, for example:

- body shell part/model identification
- the identification code and channel assignments of each displacement sensor
- the longitudinal and lateral coordinates of each sensor
- the body component (roof, underframe, valance etc.) to which each sensor was assigned
- whether each displacement reading would be carried out electronically or manually
- the number of load steps to be employed during the test.

Once all of the configuration data had been defined, it was saved on the PC's hard disk and a file of configuration information would then be downloaded to the handheld keypad in order to provide a template for manual data entry.

The data-acquisition process itself was quite straightforward. Various torque values would be applied in a series of increasing or decreasing steps. The applied torque was monitored on a digital display (updated three times per second) until each desired level of torque was obtained. At this point the applied torque would be held at a constant value and the operator would commence acquisition on all displacement channels by means of a single keystroke. The data logger returned a stream of unscaled readings in true binary format, and these were scaled by the PC's software to give displacement readings in mm. In fact, to reduce the effect of random noise, the LVDTs were scanned eight times and an average reading was obtained for each transducer. The operator would then record all of the dial gauge readings on the keypad before proceeding to apply the next torque value.

At the end of the test, the software would combine the readings acquired via the data logger with those recorded on the keypad and would then sort them according to the longitudinal coordinate of

the corresponding sensor. Finally, the data were packaged into a comma-delimited ASCII file and loaded into a spreadsheet program (Lotus 123). Specially written worksheets and macros were provided for the test engineers to facilitate analysis and plotting of the data.

11.5 Winch testing system

This is a simple example of low speed, but real-time, data-acquisition employing the technique of simultaneous sample and hold. Digital I/O channels are used to interface to external apparatus.

Overview

The client manufactured winches for various automotive functions such as manipulating spare tyres and wheels on trucks. A data-acquisition system was required for checking the performance and structural integrity of each winch.

During the test procedure, load cells were used to monitor the load developed at various points on the winch mounting and rollers, and to measure the torque applied by the winch mechanism. The speed of rotation was measured using an optical encoder coupled to a conditioning circuit that produced a DC output in proportion to the rotational speed. There were seven channels in total and it was required to sample each channel ten times per second and to provide a real-time display of the sampled data on the PC's screen.

Reconstruction accuracy

The maximum fractional rate of change of the signals to be measured was specified as 250 per cent of full scale per second, which was equivalent to a frequency of approximately 0.8 Hz. However, the average rate of change was likely to be closer to 10 per cent of full scale per second (=0.03 Hz). As the data was to be displayed and interpreted graphically, it was appropriate in this case to estimate the average accuracy inherent in signal reconstruction using the first order reconstruction equation – i.e. linear interpolation between points (see Chapter 2). On this basis, a sampling rate of 10 samples/s (per channel) was selected. This yields a 1 per cent average reconstruction error at the maximum signal frequency and about 0.002 per cent error at the average signal frequency. Both figures compared well with the specified accuracy requirements.

Data-acquisition hardware

The sensors were connected to a high speed signal-conditioning unit possessing simultaneous sample-and-hold circuitry and slots for up to eight single-channel conditioning modules. The (differential) conditioned outputs from this unit were fed to an eight-channel multiplexed ADC card with 12-bit resolution. The ADC possessed a total non-linearity better than 1 LSB and contributed a negligible inaccuracy to the readings. The ADC card's fixed-gain instrumentation amplifier was able to settle to better than 1 LSB accuracy within 10 μs of a multiplexer channel change, and the ADC conversion time was 30 μs. Both of these figures could be easily accommodated while maintaining the required sampling rate and dynamic accuracy.

The simultaneous sample-and-hold (SSH) facility of the signal-conditioning unit was controlled by a TTL-level signal generated via one of four digital output lines provided on the ADC card. The SSH circuit possessed an acquisition time of 10 μs (to an accuracy 0.01 per cent) and a settling time of 2 μs. Again, these figures did not impose any undue limitations on the sampling rate.

Only seven of the eight available channels read data from sensors. The eighth channel carried an excitation reference voltage from the signal-conditioning unit and this allowed the software to correct the load readings for small excitation drifts caused by temperature variations etc.

A number of digital input and output channels were provided for optional interfacing to a control panel, indicator lamps and motor-control apparatus. An eight-channel optically isolated digital input card allowed external equipment to control (i.e. start or abort) the test, and a 16-channel relay output card was used to signal test-status information. The optically isolated inputs provided a degree of noise immunity, but imposed a lower limit of about 1 ms on the duration of detectable digital pulses. The relay switching time was 500 μs. Both of these figures were negligible compared with specified performance requirements.

The PC itself was based on the 33 MHz 80486 DX processor and was equipped with 8 MB of RAM, a 170 MB hard disk drive and a VESA SVGA video system.

Test procedure

In preparation for the test, the operator would configure the software, defining parameters such as the file name for logging of test data, the test title and the duration of the test (up to 200 s).

The test would then be started either manually from the keyboard or automatically via one of the digital input lines. Automatic operation was facilitated by a handshaking sequence involving PC-controlled relays designated as ‘Ready to begin’ and ‘Test in progress’. Both of these relays were assigned to operate in failsafe mode: closed contacts indicated their active state. When power was removed (i.e. the PC is switched off) the contacts return to their inactive (open) state.

During the test, the channels were scanned ten times per second and the acquired data was displayed on the screen in both graphical and digital format. The various channels were colour coded for clarity. Also shown were upper and lower limits that could be applied to selected channels. These limits could be adjusted manually prior to commencing the test and were used to control software comparators and associated relays. Again these relays operated in failsafe mode: a closed contact indicating that the signal was within a specified tolerance band. When power was removed, the contacts would open indicating an out-of-tolerance condition.

At the end of the test, the acquired data would be logged (in seven-column space-delimited ASCII format) to a user specified disk file or could be downloaded to a printer for permanent storage.

Software

As the user interface requirements were quite modest, the facilities offered by Windows were outweighed by the greater degree of determinism and easy control over I/O possible with MS-DOS version 5. The software was written using the Borland Pascal 7 compiler and assembly language.

The sampling itself was interrupt based. The PC’s system timer was reprogrammed to generate interrupts 20 times per second. A new interrupt 08h routine (written in assembly language) was installed to handle the interrupts and care was taken to call the original BIOS handler at the correct average rate (18.2 Hz). As this interrupt has the highest priority, it is suitable for performing certain time-critical tasks provided, of course, that lower priority handlers and the main program thread do not disable interrupts for a significant length of time.

On every second timer interrupt (i.e. every 100 ms), the ADC was commanded to sample each channel and the readings, and then scaled, corrected for excitation drift (if appropriate) and stored in a FIFO buffer. The buffer provided a degree of decoupling between the interrupt handler and main program thread, allowing the latter to perform the relatively time-consuming task of displaying the data

on screen. As the timing of the limit-relay signals was not critical (a delay of up to 2 s was acceptable in this case) the task of updating the relay outputs was delegated to the main program thread. Obviously, in applications where a more rapid (and deterministic) response is required, interrupt-based I/O might be more appropriate.

11.6 Brake actuator test system

The requirement to dynamically measure the load vs. displacement characteristic of a component under test is common in manufacturing industries. This example uses an external timer to pace the load/displacement sampling sequence at 100 Hz.

Overview

As part of a quality control programme, a manufacturer of high performance brake actuators manually tested every assembly coming off the production line. The actuator was placed into a test jig and its piston was moved using a hand-operated screw drive, as illustrated in Figure 11.4. The resistance to motion offered by the piston arose from the combined action of a spring and friction bush and would vary throughout the test as a function of axial displacement.

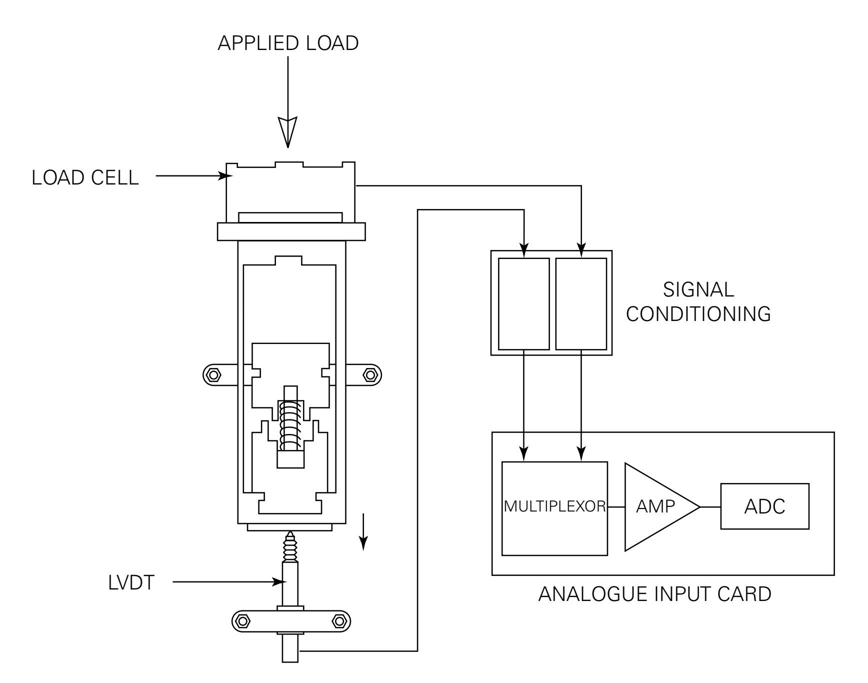


Figure 11.4 *Brake actuator test apparatus*

A data-acquisition system was required to record the load (resistance to motion) vs. displacement characteristic of the actuator and to display this graphically at the end of each test. The load/displacement curve was expected to pass through four distinct regimes, characterized by the action of the spring, friction bush and other elements of the actuator. The loads at various points on the curve and the displacements at which one regime gave way to the next represented critical design parameters.

Hardware

The sensors used consisted of an LVDT with a full-scale range of 10 mm, and an 8000 N load cell, linked to single-channel AC and DC signal-conditioning modules respectively. The latter provided the appropriate sensor-excitation supply but did not include facilities for excitation monitoring. The conditioned signals were fed to differential inputs of an eight-channel multiplexed ADC card placed in one of the PC's ISA expansion slots. The ADC card also provided a timer/counter circuit, which could be configured to trigger ADC conversions and to generate interrupts within the PC. The hardware did not provide simultaneous sample-and-hold capabilities so the potential delay between sampling of the load and displacement channels was of some concern.

The PC used was a 25 MHz 80386 unit, with numeric coprocessor, 4 MB of RAM and a 90 MB hard disk. The system was designed to operate under Microsoft MS-DOS and the software was written in a combination of C++ and assembly language.

Dynamic accuracy

The maximum permissible errors specified for the load and displacement channels were ±1.0 per cent and ±0.25 per cent of full scale respectively.

The ADC card provided 12-bit resolution with a total non-linearity better than 1 LSB (i.e. accuracy of ±0.025 per cent of full scale). The load cell and LVDT signal-conditioning units were of a high quality and contributed comparatively small non-linearities and temperature coefficients. The principal sources of inaccuracy in the measurement system arose from the non-linearities of the sensors themselves (±0.1 per cent for the LVDT and ±0.5 per cent for the load cell) and from the effects of dynamic sampling.

The maximum rate of change of the displacement signal was specified as 50 per cent of full scale per second, which is equivalent to a maximum frequency component in the signal of about 0.16 Hz.

The corresponding figures for the load signal were somewhat greater, being 1000 per cent of full scale per second and 3.2 Hz.

The aperture time of the sample-and-hold amplifier on the front end of the ADC was specified as 2 μs to an accuracy of 1 LSB. This enabled the ADC to sample signal frequencies up to around 60 Hz to ±1 LSB – considerably higher than either of the maximum signal frequencies.

The interchannel slew error was of greater concern, however. As a simultaneous sample-and-hold circuit was not available, the slew error was determined by the speed at which the system could sample the load and displacement channels. The settling time of the ADC card's instrumentation amplifier (10 μs for 1 LSB accuracy) and the ADC conversion time (25 μs) were not limiting factors. The sampling routine, which was part of an interrupt handler, was written in assembly language and, in the context of this application, it was possible to consistently obtain an interchannel slew time of less than 100 μs and an associated slew error of ±0.15 per cent.

Sampling rate

In order to allow the data to be unambiguously interpreted from its graphical representation on screen, it was necessary to sample at a sufficiently high rate. The average error involved in visually interpolating between points was approximated by the first order reconstruction equation presented in Chapter 2. This showed that an acquisition rate of at least 32 load samples per second would be required in order to maintain an average reconstruction error of 2.0 per cent of full scale. Clearly, this exceeds the specified accuracy figures, but because the PC would not make decisions or issue control signals on the basis of the reconstructed signal, this degree of error was acceptable.

An *upper* limit on the number of samples that could usefully be obtained per second was imposed by the bandwidths of the signal-conditioning units. These were quoted as 500 Hz for the LVDT conditioner and 200 Hz for the strain-gauge-bridge (load cell) unit.

An intermediate sampling rate of 100 Hz per channel was selected. This rate, rather than the lower limit of 32 samples/s, was chosen so as to facilitate the addition of a moderate degree of filtration should this be subsequently required.

Test sequence and sampling

The test was started and stopped via the keyboard. Because data was to be recorded in an internal memory buffer of limited size, the test

duration was limited to 120 s. During the test, the load and displacement signals were sampled within an interrupt handler. An 8254 timer/counter IC provided on the ADC card was programmed to generate interrupts on IRQ3 at 100 Hz (the COM2 serial port, which usually uses IRQ3, was not fitted in this instance). Within the interrupt handler, the software first obtained one load reading and then one displacement reading from the ADC. The interrupt code was written in assembly language in order to minimize interchannel slew.

A potential problem with using IRQ3 for data acquisition is that it has a lower priority than some other hardware interrupts in the system (e.g. system timer and keyboard). Interrupt processing could be temporarily and unpredictably blocked if the processor happened to be responding to one of the higher priority interrupts at the time that IRQ3 was asserted. However, the maximum variability in interrupt latency was assessed to be just a few hundred microseconds. This was within acceptable limits provided that it did not affect the interchannel slew time. The latter possibility was circumvented by careful control of processor interrupts within the IRQ3 handler.

11.7 Monitoring of bush-insertion load

This example is very similar to the brake actuator test system in that almost identical sampling techniques are used to measure load and displacement. Additional features in this example include a machine-control interface implemented via an array of relays and PLC, and pass/fail component testing.

Overview

The client used a twin-ram hydraulic press to insert bushes into circular apertures in car suspension arms. Each hydraulic ram performed an identical function: two rams simply allowed twice as many components to be processed. Once inserted into the suspension arm, each bush was held in place by friction (assisted by a shallow recess around the rim of the bush), and consequently the insertion load was an important indicator of the integrity of the assembly. Too low a load would denote a loose fit; too great a load might result from an obstructed aperture or defective bush.

It was necessary to devise a data-acquisition system for monitoring the load vs. displacement characteristics of the bush-insertion process. The client required the data to be monitored for both hydraulic rams independently. In each case the data was to be compared against upper and lower tolerance curves in order that components with improperly seated bushes could be rejected. The

load vs. displacement data would then be recorded on a batch-by-batch basis for quality control purposes. An additional requirement was that the software had to interface, via an array of relays, to a PLC that was used to control the operation of the press.

Hardware

Each hydraulic ram was fitted with a 250 kgf load cell and 200 mm LVDT having linearities of 0.5 per cent and 0.25 per cent respectively (see Figure 11.5). The sensor signals were conditioned using an eight-slot unit fitted with four single-channel signal-conditioning cards. The LVDT conditioning cards had bandwidths of 800 Hz (to −3 dB) and those for the load cells had bandwidths of 500 Hz (to −3 dB). The conditioning rack was equipped with a simultaneous sample-and-hold circuit and a facility for monitoring excitation reference voltages. The conditioned signals were digitized with a 12-bit ADC mounted on a plug-in card inside the PC. The ADC card had eight (differential) multiplexed inputs and exhibited a linearity of ±1 LSB, an aperture time of 8 μs to 1 LSB, an instrumentation amplifier settling time of 20 μs to 1 LSB and a conversion time of 35 μs.

The PC was a 16 MHz 80286 unit, equipped with 1 MB RAM and a 40 MB hard disk drive. The software was written in a mixture of Pascal and assembly language and ran under Microsoft MS-DOS version 3.3.

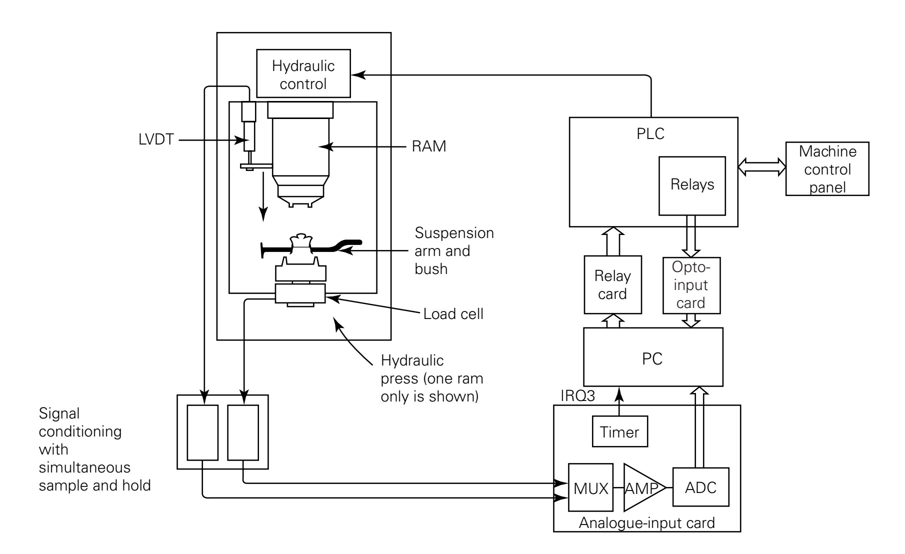


Figure 11.5 *PC-based control and monitoring system for a bush-insertion machine*

Dynamic accuracy and sampling

In terms of software design, this application has many similarities with the brake actuator test system. Load and displacement are again monitored at a rate of 100 samples/s (on each channel) using an almost identical technique. For this reason we will not discuss the dynamic analysis or interrupt-based sampling technique again, except to note that a simultaneous sample-and-hold circuit was used in this instance in order to minimize interchannel slew.

Press control functions

The PC was housed in a locked industrial enclosure, sealed to IP65. Although the screen was visible to the press operator, the keyboard could be accessed only by the production-line supervisor. During normal operation, therefore, the software could accept commands only via the machine's control panel and PLC interface. Control commands issued by the PLC consisted of digital pulses on specific opto-isolated input lines. The relay card allowed the PC to send level-sensitive status information back to the PLC.

A simple handshaking sequence was devised to synchronize the PC software with the operation of the press. When ready to begin monitoring, the PC would activate a 'Ready' relay. When the hydraulic press was ready (i.e. components in place; rams at start position; safety guard closed), the PLC would generate a 240 ms pulse on one of the PC's opto-isolated inputs. This would cause the data-acquisition software to begin monitoring the displacement channels. At the same time, the PC would issue a second relay signal to indicate that the hydraulic rams could begin to descend (see the safety note below).

The software would monitor the rams' displacement as they passed a sequence of user-specified trip levels. Each level was assigned an individual channel on the PC's relay card. The relay contacts were closed on the rams' down stroke as the measured displacement fell below each trip level in turn. They opened again in reverse order as the rams returned to their start positions. In determining the optimum settings for these trip levels, careful attention was paid to the scan time of the PLC (40 ms in this instance). The hydraulic ram could move a considerable distance in the time taken by the PLC to recognize that a trip level had been reached and this had to be accounted for in setting the trip levels.

The control functions were implemented in the software as part of the sampling algorithm – i.e. within the same interrupt handler. Each ADC reading increased monotonically (and linearly) with the corresponding measurand, and this allowed scaling of the

data to be deferred. Instead, each displacement trip level was converted to the corresponding ADC value *prior* to commencing the sampling cycle, permitting the interrupt handler to manipulate and compare unscaled data (i.e. ADC counts) using relatively fast integer arithmetic.

Additional trip levels were defined for starting and stopping the load-sampling sequence. As these were used internally by the software, no corresponding relay signals were generated. During the sampling sequence, the acquired data was plotted on twin load vs. displacement graphs. The plotting algorithm was implemented in the main program thread and decoupled from the input and output functions in the interrupt handler by means of a FIFO buffer. At the end of sampling, the recorded data points were compared against load tolerance curves and a pass/fail signal was transmitted to the PLC via the PC's relay card. The PLC and press used this signal to mark every component passing the test with a dot of paint.

Safety notes

1. The polarity of the relay signals was chosen in relation to their power-off state and to the logic of the PC–PLC handshaking sequence in order to achieve fail-safe operation.
2. It is unsafe to entrust control of potentially hazardous machinery such as a hydraulic press to PC-based software. For this reason, mechanical interlocks were used to prevent ram activation until the machine's safety guard had been closed.

11.8 Laboratory furnace temperature control

A simple example illustrating thermocouple cold junction compensation, linearization and discontinuous control techniques.

Overview

Fission tracks are microscopic trails of radiation-induced damage in the crystal lattice of geological minerals. Elevated temperatures tend to modify their structure, and thermal studies of fission-track-bearing minerals are employed to infer the thermal history of rocks in the oil exploration industry.

Thermal stability studies of fission tracks have been carried out in the laboratory by heating samples at constant temperatures for a variety of fixed time intervals. The client required a means of automatically applying more complex temperature profiles, the results of

which could be compared with conventional isothermal annealing experiments. The heating episodes were required to last from 15 minutes (isothermal) up to several weeks and would span temperatures from 50°C to 550°C, although most experiments would require temperatures in the range 200–450°C.

Hardware

The samples were heated in a Gallenkamp Tube Furnace. The furnace's manual temperature control circuitry was adapted for interfacing to a PC. Its electrical heating element was controlled by a power relay which was in turn switched by a low current relay on an 8-bit digital output card in one of the PC's expansion slots. The PC was an IBM AT model running at 8 MHz and equipped with an EGA display and 20 MB hard disk drive.

The sample temperature was sensed using a type K thermocouple, connected to a low noise thermocouple amplifier. The amplified signal was fed to one differential input of an eight-channel, multiplexed ADC card. A second channel received the output from a semiconductor temperature sensor, which was placed in close proximity to the thermocouple's reference (cold) junction. The ADC possessed a 12-bit resolution and a total non-linearity of 4 LSB.

Software

The software, written in IBM compiled BASIC, allowed the experimenter to specify, in tabular format, the temperature profile required. This consisted of a series of up to 20 isothermal episodes, linear heating episodes and exponential cooling episodes. The heating and cooling rates specified were checked against the predetermined maximum heating and cooling rates that could be obtained with the furnace, and any unattainable settings were notified to the experimenter before the heating sequence began.

Throughout the heating sequence, the software displayed the current heating step and provided a continuous digital indication of the sample temperature (thermocouple reading). In addition, any temperature excursions outside a user-specified band were indicated on the screen.

Sampling and control

The thermocouple signal and reference-junction temperature were sampled approximately nine times per second. To minimize noise, groups of 16 consecutive readings were averaged. After compensating

for the reference-junction temperature (see Chapter 3), the thermocouple signal was linearized using a 12th order polynomial. Inaccuracies in the temperature measurement arose from the thermocouple and amplifier (±1°C), the semiconductor temperature sensor (±0.5°C), the combined error of both ADC channels (±1.5°C), accuracy of the cold-junction-compensation parameters (±0.1°C), and the accuracy of the linearizing polynomial (±0.2°C). The total uncertainty in the temperature measurement was thus ±3.3°C.

The measurand was used to regulate the furnace's temperature using a discontinuous (on/off) control technique. When the temperature reading reached the desired temperature plus 2°C the heater relay was deactivated. When the temperature fell to 2°C below the desired temperature, the heater element was switched on again. In this way the temperature cycled in a narrow band about the desired level. The thermal inertia of the furnace introduced a degree of overshoot and it was found that the sample temperature could be confined to a band of width ±3°C about the desired setting.

11.9 Thermoluminescence spectrometry

This example illustrates how, in a real-time data-capture application, much of the burden of time-critical I/O can be off loaded to dedicated control and interfacing hardware, allowing data-acquisition software to run under the largely non-deterministic Windows operating system.

Background

Thermoluminescence (TL) is a phenomenon exhibited by crystalline media that have been subjected to a field of ionizing radiation. It is used for radiation dosimetry and to study the thermal history of archaeological, geological and meteoritic material. Radiation incident upon a crystalline medium will tend to displace electrons within the crystal lattice to so-called trap sites, where they may remain for long periods (up to thousands or even millions of years). Heating episodes – either natural or induced in the laboratory – allow some of the trapped electrons to return to their normal sites, releasing their stored energy in the form of visible light. The temperature at which this occurs provides researchers with information about the traps, and the wavelength (colour) of the TL emissions indicates the nature of the luminescence centres within the crystal.

Overview

The client wished to construct a PC-based system for determining TL intensity as a function of temperature and wavelength. Figure 11.6

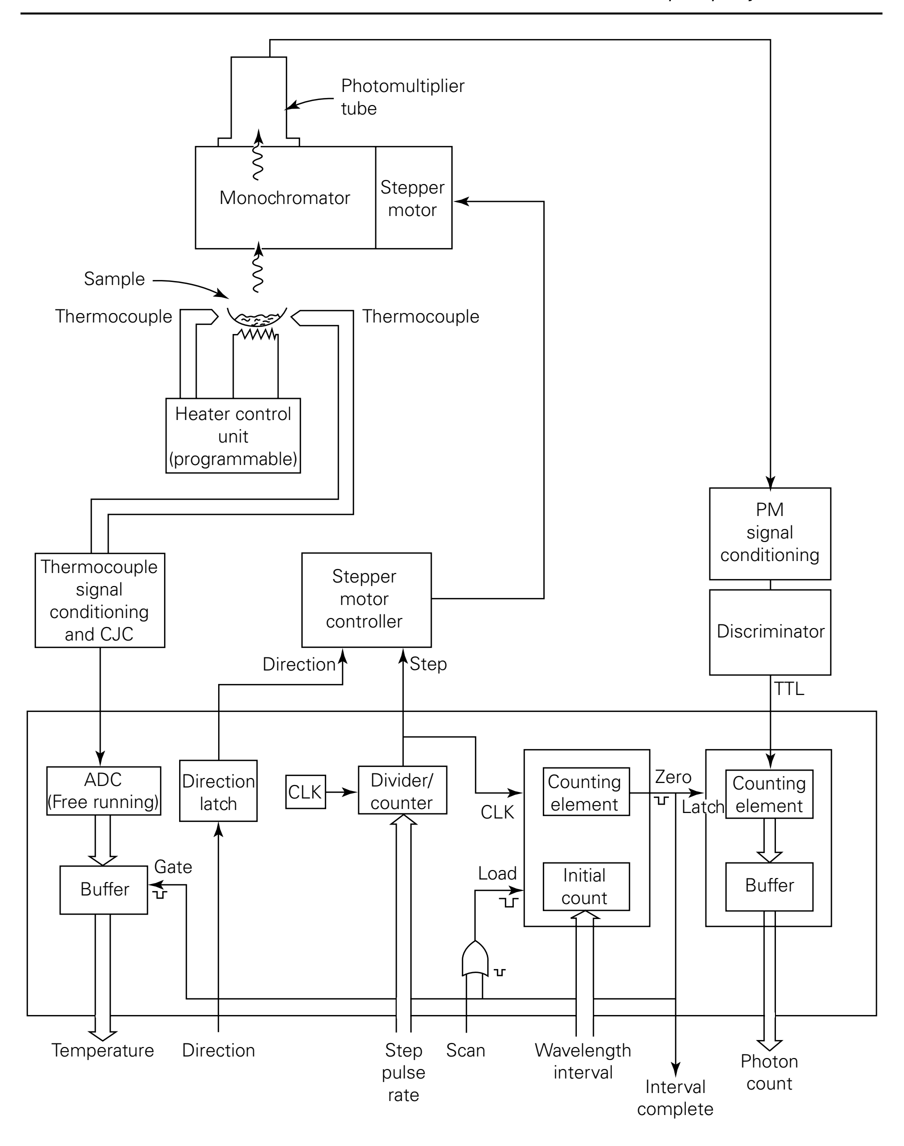


Figure 11.6 *Schematic representation of a PC-based data-acquisition system for a thermoluminescence spectrometer*

is a schematic illustration of the equipment used. The sample is heated in a partial vacuum to prevent oxidation. A dedicated heater control unit is programmed with the desired heating rate and maximum temperature. This unit then controls the temperature of the sample, using feedback from a sensing thermocouple, during the test. A second type K thermocouple supplies temperature

information to the PC via suitable signal-conditioning and cold-junction-compensation circuitry.

As the temperature is gradually increased, the sample emits light in a series of 'glow peaks', and the spectrum of the light is measured by means of a grating monochromator. This allows only light that falls within a selected narrow band of wavelengths to enter the aperture of a sensitive photomultiplier. The photomultiplier is operated in photon-counting mode and individual photon pulses are conditioned and passed through a high speed ECL discriminator and converted to TTL-level pulses.

The wavelength band transmitted by the monochromator is determined by the orientation of its diffraction grating, which is controlled using a stepper motor and associated driving circuit. The rate of wavelength change is determined by a programmable counter, which divides the clock rate down to the required stepping speed. The extent of motion of the monochromator's grating is controlled by a second counter. The programmed count represents the change in wavelength required (usually the wavelength increment between successive readings). While a scan bit is held in an active state, the counter automatically reloads and resumes counting when it reaches zero. The transition through zero generates a digital pulse that serves two functions. First, it latches the photon pulse count and thermocouple ADC readings into 24-bit and 10-bit buffers, zeroing the photon-pulse counter in the process. Second, it indicates to the PC's software that the monochromator has moved to the required wavelength. The PC uses this signal as a trigger to read the photon pulse count and thermocouple readings from the buffers. This action proceeds while the monochromator is moving on to the next wavelength.

By performing the time-critical portions of the control sequence in hardware, some of the burden of real-time operation is removed from the PC and this allows a non-deterministic operating system to be employed. The instrument-control program was designed to run under Microsoft Windows NT 4 on a 266 MHz Pentium II-based PC. It was written using an ANSI C compiler and the National Instruments LabWindows/CVI development environment.

Software facilities

After configuring the software for the required wavelength range and scanning rate, the experimenter was required to commence wavelength scanning by means of a single keystroke. At the same moment, the heater control unit's ramp generator would be started, also by manual means. Small timing errors introduced by this manual

synchronization were unimportant because the PC measured the temperature of the sample independently each time a photon count was obtained.

Throughout the test, the monochromator scanned many times through all wavelengths of interest. During each scan the software recorded typically 20 to 50 sets of temperature, light intensity (photon count) and wavelength readings. The total number of readings obtained per test was usually of the order of several hundred to two thousand, depending upon the range of experimental variables selected.

The software automatically interpolated between readings to obtain *average* temperature and wavelength values over each measurement interval. It also possessed facilities for interpolating between the skewed matrix of readings on the temperature–wavelength plane in order to provide either temperature-independent or wavelength-independent subsets of the data.

Calibration and measurement accuracy

The thermocouple signal was linearized using a 12th order polynomial to an accuracy of ±0.2°C. Other sources of inaccuracy were the electronic cold-junction-compensation module (±0.5°C) and the thermocouple itself (±1°C). The thermocouple signal was amplified such that the ADC input range would encompass temperatures from 0°C to just over 360°C. At a 10-bit resolution, this corresponds to a quantization error of ±0.2°C. ADC linearity errors were negligible. These sources introduced a total uncertainty in the temperature measurement of just under ±2°C.

Calibration of the optical system was more problematic. The photocathode of the photomultiplier tube, the monochromator's diffraction grating and other optical components exhibit wavelength-dependent transmission efficiencies. In many cases, the transmission curves do not vary smoothly with wavelength over the entire visible range (350–700 nm) and so a look-up table was constructed, into which the experimenter could load transmission-efficiency factors. These would either be measured directly using calibrated optical sources or be derived from manufacturer's specifications.

An additional correction was necessary because the finite width of the TTL pulses generated by the photomultiplier's discriminator unit meant that if two or more photons arrived within one TTL pulse cycle (typically 1 μs) they would generate only a single TTL pulse. This so-called dead time is problematic at very high pulse rates and statistical correction techniques, based on the proportion of total

detection time occupied by dead time, were applied automatically by the software.

The software allowed the experimenter to select subsets of the data and to record them in comma-delimited ASCII format on the PC's hard disk. Commercial spreadsheet programs and specialized analysis software were then used for data reduction and graphing.

Part 6 Appendices

This Page Intentionally Left Blank

Appendix A Adaptor installation reference

When installing data-acquisition cards or communications adaptors within the PC, it is usual to have to configure the card's I/O port base address, and other settings. This is normally accomplished by means of DIP switches, jumpers or installation software. To avoid memory, I/O or interrupt conflicts, these settings should be different from those chosen for other adaptor cards. Reference tables are provided in this appendix to aid in card configuration.

It is clearly impossible to list the settings used by every PC interface card on the market. Instead, only information relating to standard PC configurations and some of the more common options is shown. You should bear in mind that the information provided here is for guidance only. The assignments and addresses used in some machines may vary in certain respects from those listed here. In addition, equipment already installed in the PC might occupy the IRQ levels, DMA channels or memory and I/O addresses that appear as unused in the following tables.

Table A.1 *DMA channel assignments*

Channel	*Bits*	*PC*	*XT*	*AT and EISA*	*PS/2*
0	8	DRAM refresh	Unused	Unused[1]	Unused
1	8	Unused[2]	Unused[2]	Unused[2]	Unused[2]
2	8	Diskette	Diskette	Diskette	Diskette
3	8	Hard disk	Hard disk	Unused	Unused
4	16	Not available	Not available	Cascade DMA1	Cascade DMA1
5	16	Not available	Not available	Unused	Hard disk
6	16	Not available	Not available	Unused	Unused
7	16	Not available	Not available	Unused	Unused

Notes:

1. DMA channel 0 may be unavailable on some AT clones.
2. DMA channel 1 may be used for an SDLC serial port, if installed.

Table A.2 *Hardware interrupt (IRQ) assignments*

IRQ	*Usual vector*	*PC*	*XT*	*AT/ISA Compatible and EISA*	*PS/2*
0	08h	System timer	System timer	System timer	System timer
1	09h	Keyboard	Keyboard	Keyboard	Keyboard
2	0Ah	LPT2	Reserved	Slave PIC	Slave PIC
3	0Bh	COM2/4	COM2/4	COM2/4	COM2/4
4	0Ch	COM1/3	COM1/3	COM1/3	COM1/3
5	0Dh	Hard disk	Hard disk	LPT2	Reserved
6	0Eh	Diskette	Diskette	Diskette	Diskette
7	0Fh	LPT1[1]	LPT1[1]	LPT1[1]	LPT1[1]
8	70h	Reserved[2]	Reserved[2]	Real-time clock	Real-time clock
9[3]	71/0Ah	Reserved[2]	Reserved[2]	Reserved	Reserved
10	72h	Reserved[2]	Reserved[2]	Reserved	Reserved
11	73h	Reserved[2]	Reserved[2]	Reserved	Reserved
12	74h	Reserved[2]	Reserved[2]	Reserved[4]	Pointing device
13	75h	Reserved[2]	Reserved[2]	Coprocessor	Coprocessor
14	76h	Reserved[2]	Reserved[2]	Hard disk	Hard disk
15	77h	Reserved[2]	Reserved[2]	Reserved	Reserved

Notes:

1. Not used by BIOS. LPT1 interrupt is often disabled. IRQ7 is also generated if an unknown interrupt, caused, for example, by noise on any of the IRQ lines, is detected.
2. IRQ is not available on the PC and XT.
3. IRQ9 is software redirected on AT, PS/2 and EISA systems so that an interrupt request on this line ultimately invokes the IRQ2 handler via vector 0Ah.
4. IRQ12 is used for the pointing device (i.e. usually a PS/2 style mouse) interface on some AT clones and EISA machines.
5. On many systems, certain IRQs marked as Reserved are used by add-in adaptor cards. Other IRQs may be adopted for different purposes. IRQ3, for example, may be allocated to a network adaptor card if COM2 or COM4 is not installed. On AT, PS/2 and EISA systems, IRQ5 is commonly employed by network adaptor cards, rather than for LPT2.

Table A.3 *I/O port map for IBM PC, XT, AT and PS/2 machines*

Address range	*PC, XT*	*AT/ISA compatible and EISA*	*MCA*
000–0FFh	Used by motherboard	Used by motherboard	Used by motherboard
100–107h	Reserved for motherboard	I/O channel	POS
108–10Fh	Reserved for motherboard	I/O channel	Undocumented
110–11Fh	Reserved for motherboard	I/O channel	Undocumented
120–12Fh	Reserved for motherboard	I/O channel	Undocumented
130–13Fh	Reserved for motherboard	I/O channel	Undocumented

Table A.3 *(continued)*

Address range	*PC, XT*	*AT/ISA compatible and EISA*	*MCA*
140–14Fh	Reserved for motherboard	I/O channel	Undocumented
150–15Fh	Reserved for motherboard	I/O channel	Undocumented
160–16Fh	Reserved for motherboard	I/O channel	Undocumented
170–177h 178–17Fh	Reserved for motherboard Reserved for motherboard	Hard disk 1 Reserved	Undocumented Undocumented
180–18Fh	Reserved for motherboard	I/O channel	Undocumented
190–19Fh	Reserved for motherboard	I/O channel	Undocumented
1A0–1AFh	Reserved for motherboard	I/O channel	Undocumented
1B0–1BFh	Reserved for motherboard	I/O channel	Undocumented
1C0–1CFh	Reserved for motherboard	I/O channel	Undocumented
1D0–1DFh	Reserved for motherboard	I/O channel	Undocumented
1E0–1EFh	Reserved for motherboard	I/O channel	Undocumented
1F0–1F8h	Reserved for motherboard	Hard disk 0	Undocumented
1F9–1FFh	Reserved for motherboard	Reserved	Undocumented
200–207h 208–20Fh	Games adaptor Reserved	Games adaptor Reserved	Undocumented Undocumented
210–217h 218–21Fh	Expansion unit Reserved	Reserved Reserved	Undocumented Undocumented
220–22Fh	Reserved	I/O channel	Undocumented
230–23Fh	Reserved	I/O channel	Undocumented
240–24Fh	Reserved	I/O channel	Undocumented
250–25Fh	Undocumented	Reserved	Undocumented
260–26Fh	Undocumented	I/O channel / Reserved	Undocumented
270–277h 278–27Ah 27B–27Fh	Reserved LPT2 Reserved	Reserved LPT2 Reserved	Reserved LPT2 Reserved
280–28Fh	Undocumented	I/O channel	Undocumented
290–29Fh	Undocumented	I/O channel	Undocumented
2A0–2AFh	Undocumented	I/O channel	Undocumented
2B0–2BFh	Video subsystem (alternate)	Video subsystem (alternate)	Undocumented

Table A.3 *(continued)*

Address range	*PC, XT*	*AT/ISA compatible and EISA*	*MCA*
2C0–2CFh	Video subsystem (alternate)	Video subsystem (alternate)	Undocumented
2D0–2DFh	Video subsystem (alternate)	Video subsystem (alternate)	Undocumented
2E0h	Undocumented	Video subsystem (alternate)	Undocumented
2E1h	Undocumented	GPIB adaptor 0	Undocumented
2E2–2E3h	Undocumented	Data-acq. adaptor 0	Undocumented
2E4–2EFh	Undocumented	Reserved	Undocumented
2F0–2F7h	Reserved	Reserved	Reserved
2F8–2FFh	COM2	COM2	COM2
300–30Fh	Prototype card	Prototype card	Undocumented
310–31Fh	Prototype card	Prototype card	Undocumented
320–32Fh	Hard disk	Hard disk	Undocumented
330–33Fh	Undocumented	Reserved / I/O channel	Undocumented
340–34Fh	Undocumented	Reserved / I/O channel	Undocumented
350–35Fh	Undocumented	I/O channel	Undocumented
360–36Fh	Undocumented	Reserved	Undocumented
370–377h	Reserved 370–377h	Diskette controller	Undocumented
378–37Ah	LPT1 or LPT2	LPT1	LPT2
37B-37Fh	Reserved	Reserved	Undocumented
380–38Fh	SDLC or BSC controller 2	SDLC or BSC controller 2	Undocumented
390–39Fh	Undocumented	Cluster adaptor	Undocumented
3A0–3AFh	BSC controller 1	BSC controller 1	Undocumented
3B0–3BBh	Video subsystem	Video subsystem	Video subsystem
3BC-3BEh	LPT1 (with MDA only)	Reserved	LPT1
3BFh	Video subsystem	Video subsystem	Video subsystem
3C0–3CFh	Video subsystem	Video subsystem	Video subsystem
3D0–3DFh	Video subsystem	Video subsystem	Video subsystem
3E0–3E7h	Reserved	I/O channel	Undocumented
3E8–3EFh	Undocumented	I/O channel	Undocumented
3F0–3F7h	Diskette controller	Diskette controller	Diskette controller
3F8–3FFh	COM1	COM1	COM1

Table A.3 *(continued)*

Notes:

1. Those ports labelled ‘I/O channel’ may normally be used for DA&C cards provided, of course, that they are not already utilized by existing adaptors. Some of those addresses listed as ‘Reserved’ or ‘Undocumented’ may also be used, but you should be aware that there is a greater potential for conflicts to occur with other installed equipment. It should be remembered that many systems incorporate devices which are not listed. Network cards for instance are often located at I/O address 360h.
2. The most commonly used addresses for ADC, DAC or digital I/O cards are within the Prototype Card address range – i.e. 300h to 31Fh, although other addresses are possible. When installing additional serial communications (e.g. RS-422/485) cards on AT clones and EISA systems, it is usual to select addresses 3E8h to 3EFh for COM3 and 2E8h to 2EFh for COM4.
3. EISA machines employ an extended I/O address scheme whereby each expansion slot is allocated 1024 unique addresses. This address space is divided into four blocks of 256 contiguous I/O addresses starting at X000h, X400h, X800h and XC00h, where X is the EISA slot number: 1, 2, 3 etc. In addition to this slot-specific address space, EISA systems also incorporate the AT I/O ports which may be used with ISA compatible cards. Address ranges 400h to 4FFh, 800h to 8FFh and C00 to CFFh are also reserved for use by the EISA motherboard, although only the range 400h to 4FFh is currently used.

Table A.4 *PC, XT, AT and PS/2 conventional-memory map*

From	*To*	*Size*	*Description*
00000h	003FFFh	1K	Interrupt vector table
00400h	004FFh	256 bytes	BIOS Data Area
00500h	9FFFFh[1]	638.75K	DOS & BIOS data; DOS; DOS drivers; transient program area
A0000h	BFFFFh	128K	Display adaptor video buffers
C0000h	C7FFFh	32K	Video adaptor ROM
C8000h	DFFFFh	96K	Non-video ROM expansion
E0000h	EFFFFh	64K	Reserved for system ROM expansion (used by system ROM on MCA)
F0000h	FFFFFh	64K	System ROM

Note:

1. The upper limit varies. Older systems may be equipped with less than 640K conventional memory, leaving space below the 640K barrier for memory-mapped I/O devices and BIOSes.

Table A.5 *Common usage for display adaptor and ROM expansion areas*

From	*To*	*Size*	*Use*
B0000h	B7FFFh	32K	Monochrome display adaptor's video buffer
B0000h	BFFFFh	64K	Hercules monochrome graphics adaptor's video buffer
B8000h	BFFFFh	32K	CGA video buffer
A0000h	BFFFFh	128K	EGA, MCGA, VGA and SVGA video buffers
C0000h	C3FFFh	16K	EGA BIOS
C8000h	CBFFFh	32K	Hard disk BIOS (XT)
D0000h	D7FFFh	32K	Cluster adaptor BIOS
D0000h	DFFFFh	64K	LIM EMS page frame (although this may appear at other addresses)

Notes:

1. Adaptor ROM BIOSes and memory-mapped I/O devices may be mapped to any unused memory address range. Note, however, that many other installable devices may make use of the available memory space so care should be taken to avoid conflicts with existing adaptor cards. Unoccupied addresses within the range C0000h to DFFFFh should normally be used.
2. On 80386 and later PCs using DOS version 5 or subsequent releases, some of the memory areas above A0000h (i.e. between the adaptor BIOSes and buffers etc.) may have physical RAM mapped into them. These areas, known as Upper Memory Blocks (UMBs) can be used to run drivers and TSR programs. After installing a new adaptor card, it will normally be necessary to reconfigure the system software in order to remap the UMBs accordingly.

Appendix B Character codes

Computers, data-acquisition units and process-control devices generally communicate by transmitting and receiving a series of characters. Each character is, in fact, a binary number which is simply *interpreted* as a character by the receiving device. Both the transmitter and the receiver must of course agree on the numbers which will be used to represent each character.

Many character encoding schemes have been devised and some are now largely obsolete. Baudot and Transcode, for example, were compiled many years ago for telexes and paper-tape systems. The former is a 5-bit code which utilizes a special shift character to distinguish between letters and digits, while Transcode is a full 6-bit code which can represent 64 different characters without the need for a shift character.

The most popular character code currently in use is known as ASCII, standing for American Standard Code for Information Interchange. This is a 7-bit code, established by ANSI in the 1970s. It is capable of representing 128 different characters as listed in Table B.1. An extended, 8-bit version of the ASCII code, which can represent a total of 256 characters, is also in widespread use. The additional characters available in the 8-bit ASCII code are listed in Table B.2. Other 8-bit codes include EBCDIC (Extended Binary Coded Decimal Interchange Code) which is used almost exclusively in IBM mainframe systems. Although EBCDIC has 256 possible character codes many of these are unassigned. Because of its limited applicability to DA&C systems it will not be discussed here.

Virtually all character sets include a number of control codes. These are generally non-printable character codes, although some will display as special graphics characters on the PC. They are intended for text and message formatting and for controlling the receiving device. The common meanings and usage of these control codes are listed in Table B.3.

Table B.1 *The 7-bit ASCII character set*

Hex	*Dec*	*Char*	*Hex*	*Dec*	*Char*	*Hex*	*Dec*	*Char*	*Hex*	*Dec*	*Char*
00	0	^@	20	32	SP	40	64	@	60	96	'
01	1	^A	21	33	!	41	65	A	61	96	a
02	2	^B	22	34	"	42	66	B	62	98	b
03	3	^C	23	35	#	43	67	C	63	99	c
04	4	^D	24	36	$	44	68	D	64	100	d
05	5	^E	25	37	%	45	69	E	65	101	e
06	6	^F	26	38	&	46	70	F	66	102	f
07	7	^G	27	39	'	47	71	G	67	103	g
08	8	^H	28	40	(	48	72	H	68	104	h
09	9	^I	29	41	)	49	73	I	69	105	i
0A	10	^J	2A	42	*	4A	74	J	6A	106	j
0B	11	^K	2B	43	+	4B	75	K	6B	107	k
0C	12	^L	2C	44	,	4C	76	L	6C	108	l
0D	13	^M	2D	45	-	4D	77	M	6D	109	m
0E	14	^N	2E	46	.	4E	78	N	6E	110	n
0F	15	^O	2F	47	/	4F	79	O	6F	111	o
10	16	^P	30	48	0	50	80	P	70	112	p
11	17	^Q	31	49	1	51	81	Q	71	113	q
12	18	^R	32	50	2	52	82	R	72	114	r
13	19	^S	33	51	3	53	83	S	73	115	s
14	20	^T	34	52	4	54	84	T	74	116	t
15	21	^U	35	53	5	55	85	U	75	117	u
16	22	^V	36	54	6	56	86	V	76	118	v
17	23	^W	37	55	7	57	87	W	77	119	w
18	24	^X	38	56	8	58	88	X	78	120	x
19	25	^Y	39	57	9	59	89	Y	79	121	y
1A	26	^Z	3A	58	:	5A	90	Z	7A	122	z
1B	27	^[	3B	59	;	5B	91	[	7B	123	{
1C	28	^\	3C	60	<	5C	92	\	7C	124	|
1D	29	^]	3D	61	=	5D	93	]	7D	125	}
1E	30	^^	3E	62	>	5E	94	^	7E	126	~
1F	31	^_	3F	63	?	5F	95	_	7F	127	Δ

Notes:

1. The first 32 characters are defined as non-printable control characters. On the PC these characters may be entered by means of the Ctrl key (represented by '^' in the table) and the character shown, although they may display as graphics characters (i.e. happy face, card-suit symbols, arrows and other characters).
2. Depending upon the software running on the PC, the control characters may have other effects on the display such as moving to a new line or clearing the screen (also see Table B.3).

Table B.2 *Additional characters available in 8-bit ASCII*

Hex	*Dec*	*Char*	*Hex*	*Dec*	*Char*	*Hex*	*Dec*	*Char*	*Hex*	*Dec*	*Char*
80	128	Ç	A0	160	á	C0	192	└	E0	224	α
81	129	ü	A1	161	í	C1	193	┴	E1	225	β
82	130	é	A2	162	ó	C2	194	┬	E2	226	Γ
83	131	â	A3	163	ú	C3	195	├	E3	227	π
84	132	ä	A4	164	ñ	C4	196	─	E4	228	Σ
85	133	à	A5	165	Ñ	C5	197	†	E5	229	σ
86	134	å	A6	166	ª	C6	198		E6	230	μ
87	135	ç	A7	167	º	C7	199		E7	231	τ
88	136	ê	A8	168	¿	C8	200	╚	E8	232	Φ
89	137	ë	A9	169	⌐	C9	201	╔	E9	233	θ
8A	138	è	AA	170	¬	CA	202	╩	EA	234	Ω
8B	139	ï	AB	171	½	CB	203	╦	EB	235	δ
8C	140	î	AC	172	¼	CC	204	╠	EC	236	∞
8D	141	ì	AD	173	¡	CD	205	═	ED	237	ϕ
8E	142	Ä	AE	174	≪	CE	206	╬	EE	238	$\in$
8F	143	Å	AF	175	≫	CF	207	╧	EF	239	$\cap$
90	144	É	B0	176	░	D0	208	╨	F0	240	$\equiv$
91	145	æ	B1	177	▒	D1	209	╤	F1	241	$\pm$
92	146	Æ	B2	178	▓	D2	210	╥	F2	242	$\geq$
93	147	ô	B3	179	│	D3	211	╙	F3	243	$\leq$
94	148	ö	B4	180	┤	D4	212	╘	F4	244	⌠
95	149	ò	B5	181	╡	D5	213	╒	F5	245	⌡
96	150	û	B6	182	╢	D6	214	╓	F6	246	$\div$
97	151	ù	B7	183	╖	D7	215	╫	F7	247	$\approx$
98	152	ÿ	B8	184	╕	D8	216	╪	F8	248	∘
99	153	Ö	B9	185	╣	D9	217	┘	F9	249	•
9A	154	Ü	BA	186	║	DA	218	┌	FA	250	·
9B	155	¢	BB	187	╗	DB	219	█	FB	251	$\surd$
9C	156	£	BC	188	╝	DC	220	▄	FC	252	^{n}
9D	157	¥	BD	189	╜	DD	221	▌	FD	253	^{2}
9E	158	₧	BE	190	╛	DE	222	▐	FE	254	■
9F	159	*f*	BF	191	┐	DF	223	▀	FF	255	

Notes:

1. These characters are available only in 8-bit ASCII. Characters 00h to 7Fh in 8-bit ASCII are identical to the standard 7-bit ASCII characters listed in Table B.1.
2. Character FFh is a non-printing character.

Table B.3 *ASCII control codes*

Hex	*Name*	*Description*
00	NUL	Null: has no effect and contains no information; often used to pad the beginning of a transmission
01	SOH	Start of Header: identifies beginning of message header
02	STX	Start of Text: identifies beginning of text / data block; usually follows a message header and may be used to mark the end of the header
03	ETX	End of Text: identifies end of text / data block
04	EOT	End of Transmission: signals end of transmission; may also be used to terminate a communications session
05	ENQ	Enquiry: general request for status, information or identification
06	ACK	Acknowledgement: general affirmative response to queries/enquiries; receiving device may transmit ACK to indicate a data block has been received without error
07	BEL	Bell: sounds bell, buzzer or speaker on receiving equipment
08	BS	Backspace: move cursor/print position back one space on terminal
09	HT	Horizontal Tab: move cursor/print position to next tab-stop position on the current line
0A	LF	Line Feed: move cursor/print position down to next line
0B	VT	Vertical Tab: move cursor/print position down to next vertical tab line
0C	FF	Form Feed: move cursor/print position to top of next page; or eject printed page
0D	CR	Carriage Return: move cursor/print position to beginning of current line
0E	SO	Shift Out: indicates that subsequent characters with codes greater than 1Fh are not ASCII encoded; all characters with codes less than or equal to 1Fh are still interpreted as ASCII control codes
0F	SI	Shift In: all subsequent characters are ASCII encoded
10	DLE	Data Link Escape: marks escape sequences that are used to control transmissions
11	DC1	Device Control 1: application specific; often used as XON character in software flow control
12	DC2	Device Control 2: application specific
13	DC3	Device Control 3: application specific; often used as XOFF character in software flow control
14	DC4	Device Control 4: application specific
15	NAK	Negative Acknowledgement: general negative response to queries / enquiries

Table B.3 *(continued)*

Hex	*Name*	*Description*
16	SYN	Synchronous Idle: transmitted during synchronous communications to ensure synchronization
17	ETB	End of Transmission Block: indicates the end of each transmitted data block
18	CAN	Cancel: cancels previous data (usually up to the last CR character); may indicate that previous data contained errors
19	EM	End of Medium: no more medium (e.g. printer paper or tape)
1A	SUB	Substitute: used to replace a character that is known or suspected to be erroneous
1B	ESC	Escape: signifies the start of an escape sequence that is used to control devices such as printers; also used as a general 'abort' command in PC applications
1C	FS	File Separator: terminates transmitted files; usage is application-specific
1D	GS	Group Separator: terminates data blocks within files; usage is application-specific
1E	RS	Record Separator: terminates records within groups; usage is application-specific
1F	US	Unit Separator: terminates units within records; usage is application-specific
7F	DEL	Delete: deletes character at cursor position

Note:
Many systems make use of only a few of these control codes. Their usage may not always be entirely consistent with that outlined. DC1 to DC4 and FS, GS, RS and US all have application-specific meanings. Their usage will vary between different devices and protocols.

Recently, a 16-bit character encoding scheme known as Unicode has been developed as an international standard by a consortium of companies, including IBM, Microsoft and Apple. This scheme includes not only the Roman alphabet, but also Russian, Greek, Arabic, Chinese and other character sets as well as a number of mathematical symbols and punctuation marks. It is capable of representing up to 65 536 different characters in total. The first 128 Unicode characters are identical to the standard 7-bit ASCII character set. Unicode is presently used in Microsoft's Windows NT. Because of its size and complexity, it seems unlikely that Unicode will supersede ASCII in industrial communications and real-time data-acquisition systems, at least for some considerable time.

This Page Intentionally Left Blank

References

I have tried, as far as possible, to refrain from referencing original manufacturer's technical literature, which can sometimes be difficult or expensive to obtain. Instead, the majority of references are for easily accessible books: most will (hopefully) still be in print by the time that the present work is published. However, the text does contain a small number of references to manufacturers' data sheets relating to common PC components and subsystems. These can usually be obtained, individually or in the form of published data books, direct from the manufacturers concerned or from various component suppliers.

Application notes, which often accompany manufacturers' product catalogues, can be a very useful source of information. In spite of this, and because of their proprietary and sometimes transient nature, I have included only a few references to such publications in the text. Of particular note are the series of Applications Handbooks published by Burr Brown Corporation, PO Box 11400, Tucson, AZ 85734-1400, USA, and Data Translation Inc., 100 Locke Drive, Marlborough, MA 01752-1192.

Adamson M. (1990) *Small Real Time System Design: From Microcontrollers to RISC Processors.* Sigma Press.

Bannister B.R. and Whitehead D.G. (1991) *Instrumentation: Transducers and Interfacing.* Chapman & Hall.

Bell D., Morrey I.I. and Pugh J.R. (1992) *Software Engineering: A Programming Approach*, 2nd edn. Prentice-Hall International (UK) Ltd.

Ben-Ari M. (1982) *Principles of Concurrent Programming.* Prentice-Hall Inc.

Brown R. and Kyle J. (1991) *PC Interrupts: A Programmer's Reference to BIOS, DOS and Third Party Calls.* Addison-Wesley Publishing Company Inc.

Buchanan W. (1999) *PC Interfacing, Communications and Windows Programming*. Addison-Wesley Longman Limited.

Collett C.V. and Hope A.D. (1983) *Engineering Measurements*, 2nd edn. Longman Scientific and Technical.

Crozier P. (1985) *Electronic Instruments and Measurements*. Breton Publishers.

Dettmann T. and Johnson M. (1992) *DOS Programmer's Reference*, 3rd edn. Que Corporation.

Duncan R. (1988) *Advanced MS-DOS Programming*, 2nd edn. Microsoft Press.

Duncan R. (1989) *MS-DOS Extensions*. Microsoft Press.

Duncan R., Petzold C., Baker M.S., Schulman A., Davis S.R., Nelson R.P. and Moote R. (1990) *Extending DOS*. Addison-Wesley Publishing Company Inc.

Edgar T.F. (1996) Process Dynamics and Control, in *The Electronics Handbook* (J.C. Whitaker, ed.), pp. 1823–1839. CRC Press Inc.

Eggebrecht L.C. (1990) *Interfacing to the IBM Personal Computer*, 2nd edn. Howard Sams.

Evesham D.A. (1990) *Developing Real-Time Systems – A Practical Introduction*. Sigma Press.

Fröberg C.-E. (1966) *Introduction to Numerical Analysis*. Addison-Wesley Publishing Company Inc.

Grover D. (ed.) (1989) *The Protection of Computer Software – Its Technology and Applications*. Cambridge University Press.

Hogan T. (1988) *The Programmer's PC Sourcebook*. Microsoft Press.

Holzner S. and Peter Norton Computing Inc. (1991) *Advanced Assembly Language*. Brady.

Hummel R.L. (1992) *PC Magazine Programmer's Technical Reference: The Processor and Coprocessor*. Ziff-Davis Press.

IBM Corporation (1989) *PS/2 BIOS Interface Technical Reference*. IBM Corporation.

Johnson C.D. (1988) *Process Control Instrumentation Technology*, 3rd edn. Prentice-Hall Inc.

Knuth D.E. (1973) *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company Inc.

Labfacility Ltd (1987) *Temperature Sensing with Thermocouples and Resistance Thermometers: A Practical Handbook*. Labfacility Ltd, Middlesex TW11 8LR.

Lai R.S. (1987) *Writing MS-DOS Device Drivers*. Addison-Wesley Publishing Company Inc.

Maguire S.A. (1993) *Writing Solid Code*. Microsoft Press.

Maine A.C. (1986) *Interfacing Standards for Computers*. The Institution of Electrical and Electronics Incorporated Engineers.

Marnham D.J. (1994) *Interfacing Standards for Computers*, 2nd edn. The Institution of Electronics and Electrical Incorporated Engineers.

Miller A.R. (1993) *Borland Pascal Programs for Scientists and Engineers.* Sybex Inc.

Mitchell E. (1993) *Borland Pascal Developer's Guide.* Que Corporation.

Norton P. and Wilton R. (1988) *The New Peter Norton Programmer's Guide to the IBM PC and PS/2.* Microsoft Press.

Oney W. (1996) *Systems Programming for Windows 95.* Microsoft Press.

Parr E.A. (1986) *Industrial Control Handbook, Volume 1: Transducers.* Collins.

Petzold C. (1996) *Programming Windows 95.* Microsoft Press.

Phoenix Technologies Ltd. (1989) *System BIOS for IBM PC/XT/AT Computers and Compatibles.* Addison-Wesley Publishing Company Inc.

Pople J. (1979) *BSSM Strain Measurement Reference Book.* The British Society of Strain Measurement, Newcastle-upon-Tyne NE6 5QB, UK.

Press W.H., Flannery B.P., Teukolsky S.A. and Vetterling W.T. (1992) *Numerical Recipes in Pascal: The Art of Scientific Computing.* Cambridge University Press.

Putman B.W. (1987) *RS-232 Simplified – Everything You Need to Know About Connecting, Interfacing, and Troubleshooting Peripheral Devices.* Prentice-Hall Inc.

Rosch W.L. (1996) *Printer Bible.* MIS Press.

Sanchez J. and Canton M.P. (1994) *PC Programmer's Handbook*, 2nd edn. McGraw-Hill Inc.

Schulman A., Michels R.J., Kyle J., Paterson T., Maxey D. and Brown R. (1990) *Undocumented DOS.* Addison-Wesley Publishing Company Inc.

Solomon D.A. (1998) *Inside Windows NT*, 2nd edn. Microsoft Press.

Stallings, W. (1997) *Data and Computer Communications*, 5th edn. Prentice-Hall Inc.

Swan T. (1989) *Mastering Turbo Assembler.* Hayden Books.

Templeman J. (1998) *Beginning Windows NT Programming.* Wrox Press Ltd.

Tompkins W.J. and Webster J.G. (eds) (1988) *Interfacing Sensors to the IBM PC.* Prentice-Hall Inc.

Tooley M.H. (1992) *Data Communications Pocket Book*, 2nd edn. Butterworth-Heinemann Ltd.

Tooley M.H. (1995) *PC-based Instrumentation and Control*, 2nd edn. Butterworth-Heinemann Ltd.

van Gilluwe F. (1994) *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas.* Addison-Wesley Publishing Company Inc.

Vears R.E. (1990) *Microprocessor Interfacing*. Butterworth-Heinemann Ltd.

Wadlow T.A. (1987) *Memory Resident Programming on the IBM PC*. Addison-Wesley Publishing Company Inc.

Warring R.H. and Gibilisio S. (1985) *Fundamentals of Transducers*. Tab Books Inc.

Index